Docket No. 50325-0763 (Seq. No. 895) PATENT

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re Application of

David CHERITON	:	Confirmation No.: 8141
Serial No.: 09/655,295	:	Group Art Unit: 2665
Filed: September 5, 2000	:	Examiner: RYMAN, Daniel J.

Title: M-TRIE PLUS: EXTENDED TRIE BASED PACKET LOOKUP PROCESSING

Mail Stop RCE
Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

DECLARATION UNDER 37 C.F.R. 1.131

I, David R. CHERITON, declare:

1. I am the inventor named in the above-referenced application. Presently my occupation is Professor of Computer Science, Stanford University, Stanford, California.

2. I understand that the Office Action mailed December 29, 2004 rejects the pending claims of this application by relying in part on U.S. Patent No. 6,704,313 (Duret et al., hereinafter "DURET"), filed on January 28, 2000.

3. A functioning version of a computer program that embodies an invention disclosed and claimed in the above-referenced application (hereinafter "the subject invention") was created in the United States at a date long prior to January 28, 2000, and long prior to February 12, 1999.

4. I conceived of the subject invention while serving as TECHNICAL ADVISOR in the Gigabit Switching Group (GSG) of the ultimate owner of this application, Cisco Systems, Inc. A student-intern, who was employed by Cisco Systems, Inc. at the time, created under my supervision a functioning version of the subject invention prior to January 28, 2000 and prior to February 12, 1999.

5. I have reviewed the currently pending Claims 16, 18-26, 28-30, and 32-49 of the application, and to the best of my recollection, the working software program that was developed prior to January 28, 2000 and prior to February 12, 1999, and the written documentation for it, are for an embodiment of the invention that is within the scope of Claims 16, 28, 29, and 30.

6. Attached, as Exhibit 1, is a true and correct copy of a technical document titled "Access Control List Processing with Mtrie+". The technical document has been redacted by my attorneys to preserve trade secrets. The date of this document is redacted, but the true date is long prior to January 28, 2000 and long prior to February 12, 1999. The document describes an Mtrie Plus Engine (MPE) (page 12, sections 3.1 and 3.1.1) that performs a look-up method using an Mtrie+ data structure which is substantially the same as the structure recited in Claims 16, 28, 29, and 30. As described in the document, a data packet is received at an input interface (page 12, section 3.1.1), the Mtrie+ structure is organized as a multi-level tree (page 15, Figure-11), where each node in the tree includes and address and an oppcode (page 16, Section 3.2.1, Figure-12), and the lookup is terminated when a *Term* instruction or an output leaf is reached (page 14, numbered paragraphs 4 and 5). Further, the document shows an actual implementation of an MPE in C++ that is used in a simulation of looking up different protocol packets in different

Mtrie+ structures (Page 23, section 4.0 and 4.1; pages 24-31, section 4.3; pages 42-50, Appendix B). Finally, the document describes the results of the implemented simulation, and compares the performance of routing and/or filtering packets using the ACL-Mtrie+ data structure to the performance of devices and network elements that use other methods for routing and/or filtering packets. (Page 36, Table-5; see generally pages 32-36 and page 38, section 6). Thus, this document is hereby submitted as probative of a reduction to practice of the subject invention prior to January 28, 2000 and prior to February 12, 2005.

7. All of the acts set forth herein occurred or were performed by me or by someone under my supervision in the United States.

8. To the best of my knowledge, all of the acts set forth herein were carried out in secret and internal to our employer, Cisco Systems, Inc. To the best of my knowledge, any substantive disclosures to any parties external to our employer were protected from disclosure by non-disclosure agreements.

9. All documents and acts set forth herein in this Declaration relate to internal software development. None of the acts set forth herein involve public use, sale, offer for sale, publication, or other unprotected disclosure of the subject invention.

10. I declare that all statements made herein of my own knowledge are true, and that all statements made on information and belief are believed to be true; and further, that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of title 18 of United States Code, and that such willful false statements may jeopardize the validity of the application or any patent issuing thereon.

Executed at Palo Alto, California on the date set forth below.

(city) (state)

Prof. David R. Cheriton

Dated: Feb 2, 2005

Exhibit 1

CISCO SYSTEMS

Document Number	
Revision	
Author	Rolf Arnet
Project Manager	Andy Bechtolsheim

Access Control List Processing with Mtrie+

Project Headline

This documents describes the design issues and design considerations for access control list processing with the data structure Mtrie+.

Reviewers

Department	Name	Acceptance Date

Modification History

Rev	Date	Originator	Comment
		Rolf Arnet	

Definitions

ACL	Access Control Lists as they are used for configuring router running IOS. This lists define the accessibility of networks and hosts through this router.
CAM	Content Addressable Memory
CoS	Class of Service
QoS	Quality of Service

Abstract

Routers offer the facility to filter packets at their interfaces. Enabling packet filtering reduces the forwarding rate to 10'000 pps. That is because the ACL processing is executed sequentially and in software. The next generation routers are seeking to increase the forwarding rate dramatically to 100 mpps. Only a hardware solution can achieve a speedup of 10'000.

In this thesis we explore the new approach of how to process Access Control Lists using the data structure Mtrie+. The Mtrie+ data structure is created by software and used by the routers Mtrie Plus Engine to forward packets.

An analysis of Access Control Lists and traffic traces lead to rules for building a Mtrie+ data structure. The goal was to keep low the lookup count of a filtering decision and the memory usage of the data structure.

An analysis of ACLs and traffic traces lead to rules for building a Mtrie+ data structure. This new approach was explored with a simulation in C++. The goal of the simulation was to keep low the lookup count of a filtering decision and the memory usage of the data structure. Packet forwarding rate for expected traffic is over 100 mpps.

Environment

I performed this diploma thesis at the Gigabit Switching Group (GSG) of Cisco Systems Inc., San Jose, CA, U.S.A. Cisco Systems Inc. is the leading company on the computer networking market and has more than 10'000 employees all over the world. The headquarter is located at San Jose, CA.

The welcome at Cisco was very friendly. The people had always time to answer my questions. The also gave me the opportunity to participate on weekly meetings and other presentations.

Within Cisco I was reporting to Andy Bechtolsheim, VP of Gigabit Switching Group (GSG) as my manager. David Cheriton was my advisor.

Contents

1.0 Introduction

The growth of the internet and the corporate networks let the volume of the packet traffic continuously increase. Consequently the speed requirements of routers raise more and more. Soon the todays packet forwarding rate of about 60 kpps is outdated and is in the range of 100 mpps.

Routers offer the facility to filter packets at their inbound and outbound interfaces. These filters are defined by Access Control Lists (ACLs) at the router during setup time and may be changed during the operation. Enabling the packet filtering reduces the forwarding rate significantly to about 10 kpps. That is because the ACL processing is executed sequentially and in software.

To meet the high speed requirements of the next generation routers the ACL processing has to be improved. Just improving the software will never result in a speedup of 10'000. Only a hardware solution will get us to this high forwarding rate.

In this thesis we explore a new approach with the data structure Mtrie+, based on an idea of David Cheriton. Basically the Mtrie+ is an extension of the existing Mtrie which is a tree like data structure. The Mtrie+ is used by the Mtrie Plus Engine (MPE) to perform traffic filtering as well as to generate routing decisions. The MPE is a multi-threaded "processor" which services a queue of packet headers. Until now the routing and ACL processing was always separated. With the Mtrie+ both can be merged into one device. High speed packet header processing is achieved by the wide memory bus and by the multiple pipelined threads of the MPE.

The ACL of a router configuration file is first compiled into an ACL-Mtrie+ data structure which is located at the routers memory. Each single thread of the MPE then collects the next packet header in the queue and traverses the Mtrie+ data structure to generate the filtering and routing decision. In this tree structure each node has several links to other nodes. The threads select the right link by executing operations on the packet header. These operations are determined by the opcode of the node. When a thread reaches a leaf in the tree, its opcode advises the thread whether to drop or to forward the packet.

1.1 The Router

The next generation router is an exploratory design for a high-performance, highly integrated router chip set using shared memory implemented by multi-bank pipelined SDRAM. It is supporting several OC-48 ports, several Gigabit ethernet ports and a big number of 10/100 Mbit ethernet ports.

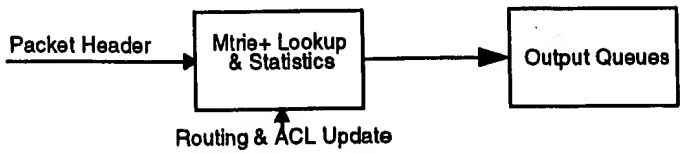

Figure-1 Fast Forwarding Pipeline

The technology used for the realization of the next generation router, is based on state-of-the-art

ASIC technology with a very high level of integration. There are very high clock frequencies and very wide memory busses in order to reduce the initial clock speed coming from the OC-48 interfaces.

1.2 About this Document

The chapter "Access Control Lists" on page 3 analyses the problems of todays packet filtering in routers. It shows their current performance limits and the impact on the packet forwarding speed. An introduction to the ACL specification language follows and its impact on the ACL-Mtrie+ data structure is discussed. Several router configuration files have been analysed in order to find general rules for the Mtrie+ design. An analysis of packet traces leads to a protocol breakdown which shows where the possible optimizations are in the Mtrie+ data structure design. A short overview of the MPE architecture shows where the Mtrie+ is located and how the packet filtering is processed.

In the chapter "Solution based on ACL Mtrie+" on page 16 the Mtrie+ data structure is described. A specification of the simulation and implementation consideration follow. Then the implementation of the ACL-Mtrie+ is illustrated.

The chapter "Evaluation" on page 32 several different ACL-Mtrie+ data structures show their performance in terms of lookup counts and memory usage. Some numbers clarify the cost of using this approach and its advantages and drawbacks. It is compared with the current IOS ACL implementation and the soon available CAM approach. Problems and further work with this data structure are discussed.

The concluding remarks are found in chapter 6.0 on page 38.

In "Appendix A: Mtrie" on page 40 the current Mtrie data structure is briefly illustrated.

In "Appendix B: Class description of the ACL-Mtrie+ Simulation" the methods and functions of the simulation are explained.

In "Appendix C: Programs for the analysis of ACLs and traces" the usage of those programs is explained.

In "Appendix D: Example in how Subdivide the Lists" the process of how the depth of a list is reduced is illustrated.

2.0 Access Control Lists

In this chapter we get first an introduction to Access Control Lists. Next we have a look on how ACL are processed currently and at the involved limitations. Finally we analyze router configurations and packet traces.

2.1 Introduction to ACL

Access control lists, or short access list, are located at routers and are also called traffic filters. ACLs allow to control whether router traffic is forwarded or dropped at the routers interfaces. The ACLs provide a basic level of security for accessing the network.

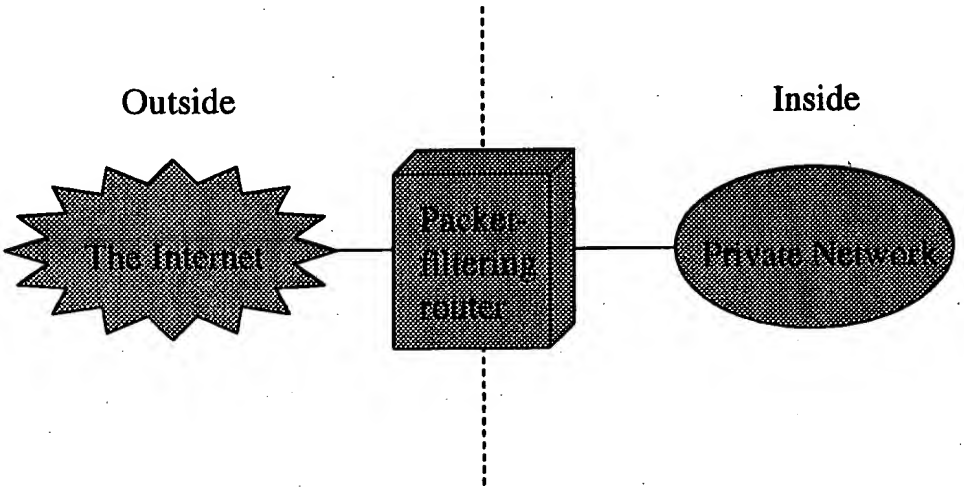

Figure-2 Packet Filtering Router

By setting up traffic filters at the router, one can control which traffic enters or leaves the own network. ACLs are commonly used in "firewalls". Typically, a router configured for traffic filtering is positioned between the internal network and an external network such as the Internet (Figure-2).

ACLs can be used for other purposes than only for controlling the transmission of packets on an interface. Controlling virtual terminal line access and restricting contents of routing updates are other possible applications.

ACLs must be defined on a per-protocol basis. For every protocol enabled on an interface of the router an ACL must be defined to control traffic flow for that protocol. All ACLs are identified by either a name or a number which is assigned when it is defined.

ACLs apply to interfaces as either inbound or outbound or both (Figure-3). If the ACL is inbound, when the router receives a packet its software checks the access list's criteria statements for a match. If the packet is permitted, the software continues to process the packet. If the packet is denied, the software discards the packet. If the ACL is outbound, after receiving and routing a packet to the outbound interface, the software checks the access list's criteria statements for a match. If the packet is permitted, the software transmits the packet. If the packet is denied, the

software discards the packet.

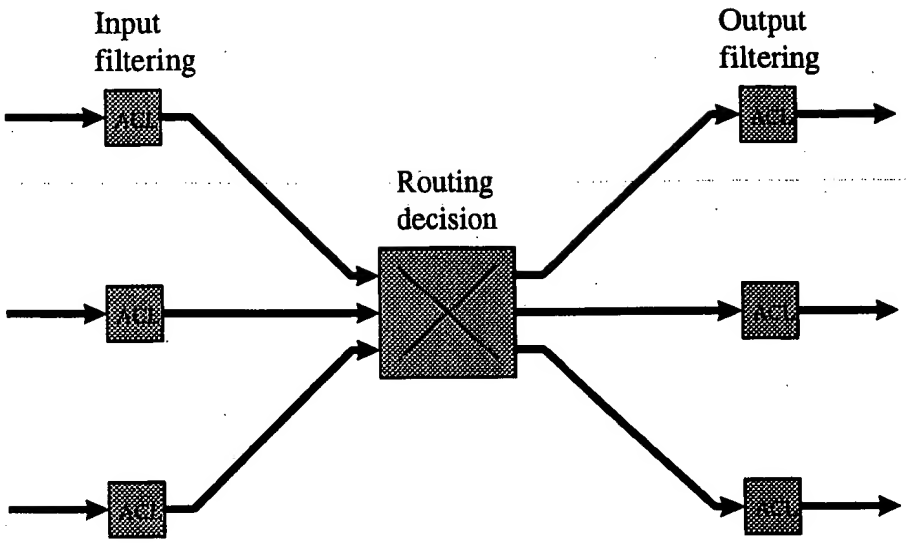

Figure-3 Packet Processing Logic

The access lists are defined in the routers configuration file. There are two forms of access list in the IOS: The standard ACL and the extended ACL. See also [1] and [2] for more detailed information.

Standard access list:

access-list access-list-number {permit | deny} {source [*source-wildcard*] | any}
access-list-number ranges from 1-99
Example: access-list 1 permit 131.108.5.17

Standard access lists are used to control traffic based on one or more source IP addresses. The IP address qualifier is a 32bit quantity in four-part, dotted-decimal format. The *wildcards* or masks are used to specify which bits of the address are compared with the packet header. Ones in the mask field specify the bit positions which are ignored, e.g. to ignore all bits set the mask or *source-wildcard* qualifier to *255.255.255.255*.

Extended access list:

access-list access-list-number [dynamic *dynamic-name* [timeout minutes]] {permit | deny} {protocol | protocol-keyword} {source *source-wildcard* | any} [operator port [port]] {destination *destination-wildcard* | any} [operator port [port]] [established] [precedence *precedence*] [tos *tos*] [log]
access-list-number for extended access list ranges from 100-199 or could be identified using a name.
Example: access-list 101 permit tcp 0.0.0.0 255.255.255.255 199.38.17.21 0.0.0.0 gt 1023
or access-list 101 permit tcp any host 199.38.17.21 gt 1023

Extended access lists provide a finer granularity in controlling the traffic. So now keyword abbreviations can be used for certain values, e.g. *any* for the mask *255.255.255.255* or *host* for the mask

0.0.0.0. The option *log* causes an informational logging message about the packet that matches the entry. The option *established* will play an important role in the design of the ACL Mtrie+.

The command *ip access-group* is used to apply an inbound or outbound access list to a specific interface:

```
interface Ethernet0
ip address 132.201.55.22 255.255.255.224
ip access-group 101 out
```

The access list is a sequential collection of permit and deny conditions that apply to the different parameters of an incoming packet. The router tests each of these parameters against the conditions in an access list one by one. The first match determines whether the router accepts or rejects the packet. Because the router stops testing conditions after the first match, the order of the conditions is critical. If no conditions match, the router rejects the packet. So the order of the access list conditions is very important and has a strong influence on the ACL Mtrie+. An example could help:

```
access-list 100 deny icmp host 129.132.3.11 132.201.0.0 0.0.255.255
access-list 100 permit icmp 0.0.0.0 255.255.255.255 132.201.0.0 0.0.255.255
```

The first line denys the forwarding of any packets from the host 129.132.3.11 to the network 132.201.0.0. The second line permits all ICMP packets sent to the same network. The router tests the two lines one by one. Therefore the semantic is "all icmp packets except packets coming from the host 129.132.3.11 are forwarded to the network 132.201.0.0". If we swap both statements the semantic is "all icmp packets are forwarded to the network 132.201.0.0" and is definitively not the same.

2.2 Current ACL Processing

Currently the Cisco IOS software uses ACLs for packet filtering. If the traffic filtering on a router is enabled, then all packets are processed by the routers cpu. Until now there is no hardware support available. How is the filtering executed?

ACL definitions provide a set of criteria which are applied to each packet that is processed by the router. The router decides whether to forward or drop each packet based on whether or not the packet matches the access list criteria.

Typical criteria defined in ACLs are packet source addresses, packet destination addresses, or upper-layer protocol of the packet. However each protocol has its own specific set of criteria that can be defined.

The order of ACL statements is important. When the router is deciding whether to forward or drop a packet, the Cisco IOS software tests the packet sequentially against each criteria statement in the order the statements were created. After a match is found, no more criteria statements are

checked. The data structure is similar to a single linked list of the statements (Figure-4).

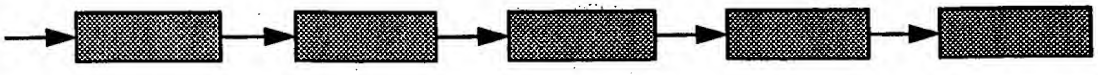

Figure-4 Data Structure of the IOS ACL

If a criteria statement explicitly permits all traffic, no later statements in the list will ever be checked. The same applies to a statement which denies all traffic. At the end of every ACL is an implied "deny all traffic" criteria statement. Therefore, if a packet does not match any of the criteria statements, the packet is dropped.

The speed of the packet forwarding degrades if the ACLs are enabled on a packet filtering router. Long ACLs (~100 statements) drops the router performance from 65kpps to around 19kpps, i.e. only 30% of the original speed is available. The reason for this severe performance impact is that every packet gets processed by the routers cpu and the enabled traffic filtering adds now an additional processing overhead.

The Internet Service Providers and other customers rely more and more on packet individual features like access lists. With the growth of the internet and the corporate networks the packet forwarding rate is raising. Soon the current speed with enabled access lists is outdated and there is the need for routers with forwarding rates around 100 mpps. Improving the software does not scale well in the necessary speedup to allow a packet forwarding rate of 100 mpps, not even with the fastest available processors.

2.3 Analysis of Router Configurations

The router configuration analysis returns statistical information about the structure of ACLs.

When we look at the ACL statements we always remember that the ACL-Mtrie+ is similar to a tree structure. This tree structure is built qualifier by qualifier of the statement. Therefore we compare the different qualifiers among the ACL statements sequentially.

A database of about 130 examples of router configurations was available. All those examples have in common to have more than 50 effective ACL rules that concerns the IP-traffic.

2.3.1 Protocol Breakdown

A total of over 44'000 statements in all 130 router configuration examples were analyzed. The most frequent protocols are TCP, UDP and IP. Furthermore there are additional qualifiers for established TCP and ICMP. The protocols IPinIP, IGMP, IPv6, GRE, EIGRP and OSPF are very rarely used. ACL statements which match any internet protocol (including ICMP, TCP and UDP) are using the keyword *ip* and therefore have somewhat more general significance. The protocols TCP and UDP both use layer 4 port addresses. As we can see in the Figure-5, their share is about 60% of all ACL statements. The high usage of layer 4 port addresses will play an important role in

the implementation of the ACL-Mtrie+.

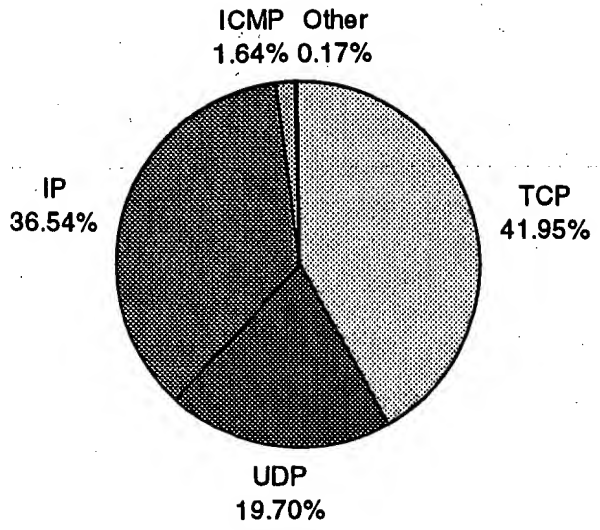

Figure-5 ACL Protocol Breakdown

Let us have a closer look at the statements with an *established* qualifier. In the Figure-6 we collected all the ACLs which use such statements. We see that 51% of those ACLs have only one statement with an *established* qualifier and this one permits all established TCP traffic.

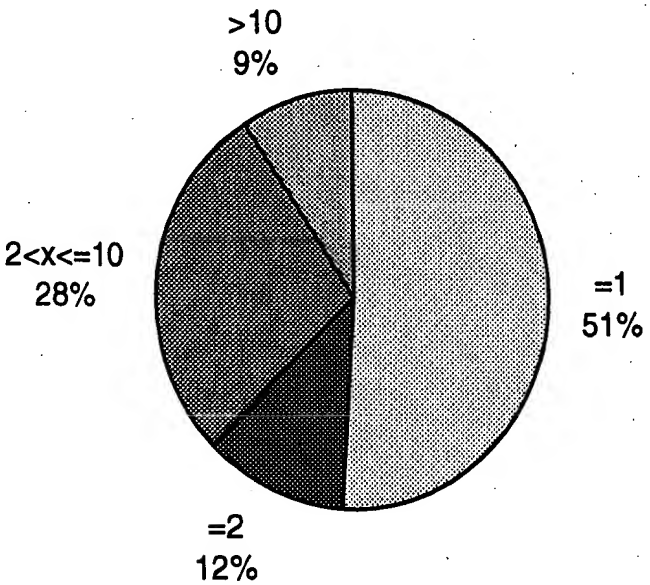

Figure-6 Established TCP Breakdown

2.3.2 IP Address

We know that the order of the statements within an ACL is important and that the IP masks define ranges of possible addresses. This combination results in an overlap in the IP address space. A

property which has an impact to the ACL-Mtrie+. A few examples could help:

```
access-list 103 permit ip 0.0.0.0 255.255.0.0 host 153.114.224.79
access-list 103 permit ip 0.0.0.0 255.255.255.255 host 153.114.224.79
```

Here the source IP addresses do overlap, but actually the second line includes the first line in its range. Hence the first line is redundant.

```
access-list 103 permit ip any any
access-list 103 deny ip host 152.163.25.3 any
```

This is an example for a wrong sequence of ACL statements. All lookups concerning this protocol will stop after the first line is processed because it matches every time. The second line is simply ignored.

```
access-list 102 permit ip 137.188.235.0 0.0.0.255 153.114.105.0 0.0.0.255
access-list 102 permit ip 137.188.239.0 0.0.0.255 153.114.105.0 0.0.0.255
```

We see that both line look similar except to the third byte in the source IP address. If we only look at the destination address, this would collapse both statements to one, i.e. both statements are placed at the same leaf position in a tree data structure.

2.3.3 Layer 4 Ports

When we browse through the ACLs, we discover that there are a lot of similar statements which have equal IP source addresses or destination addresses. The following example identifies 132.201.53.54 and 132.201.55 as a group of ftp servers:

```
access-list 100 permit tcp 199.172.91.6 0.0.0.0 132.201.53.54 0.0.0.0 eq 20
access-list 100 permit tcp 199.172.91.6 0.0.0.0 132.201.53.54 0.0.0.0 eq 21
access-list 100 permit tcp 199.172.91.6 0.0.0.0 132.201.53.55 0.0.0.0 eq 20
access-list 100 permit tcp 199.172.91.6 0.0.0.0 132.201.53.55 0.0.0.0 eq 21
access-list 100 permit tcp 199.172.90.46 0.0.0.0 132.201.53.54 0.0.0.0 eq 20
access-list 100 permit tcp 199.172.90.46 0.0.0.0 132.201.53.54 0.0.0.0 eq 21
access-list 100 permit tcp 199.172.90.46 0.0.0.0 132.201.53.55 0.0.0.0 eq 20
access-list 100 permit tcp 199.172.90.46 0.0.0.0 132.201.53.55 0.0.0.0 eq 21
```

In this example the (ftp) ports 20 and 21 appear always in ACL entries that are identical otherwise.

```
access-list 142 permit udp any host 152.163.200.2 gt 1023
access-list 142 deny   udp any any gt 33553
access-list 142 permit udp 152.163.0.0 0.0.255.255 any gt 33420
```

If we start identifying ACL statements with the port number, we see here that the layer 4 port addresses can overlap. Further let us assume that we browse first through the layer 4 port

addresses of all ACL statements and after that we look at the IP addresses. Then the first line will collapse the next two in respect to the port address. Someone could disagree and point to the fact, that the statements differ in their IP addresses. But then we falsely deny all traffic below the port number 33554 and then we accept no traffic between 33431 and 33553.

Furthermore in all the 130 router configuration examples the source port qualifier was never used, and in all over 44'000 statements in the ACL database we could only find one statement which combined a layer 4 destination port address and the qualifier established. But the qualifiers are defined in the IOS ACL specification language and therefore we can not neglect them completely. But we will take advantage of this knowledge.

Another interesting fact is that a lot of statements use the eq operator and range operator like gt, lt and range are very rarely used, except the gt 1023 rule. So we do not have to consider all possible port numbers and there is the potential to save space (Figure-7).

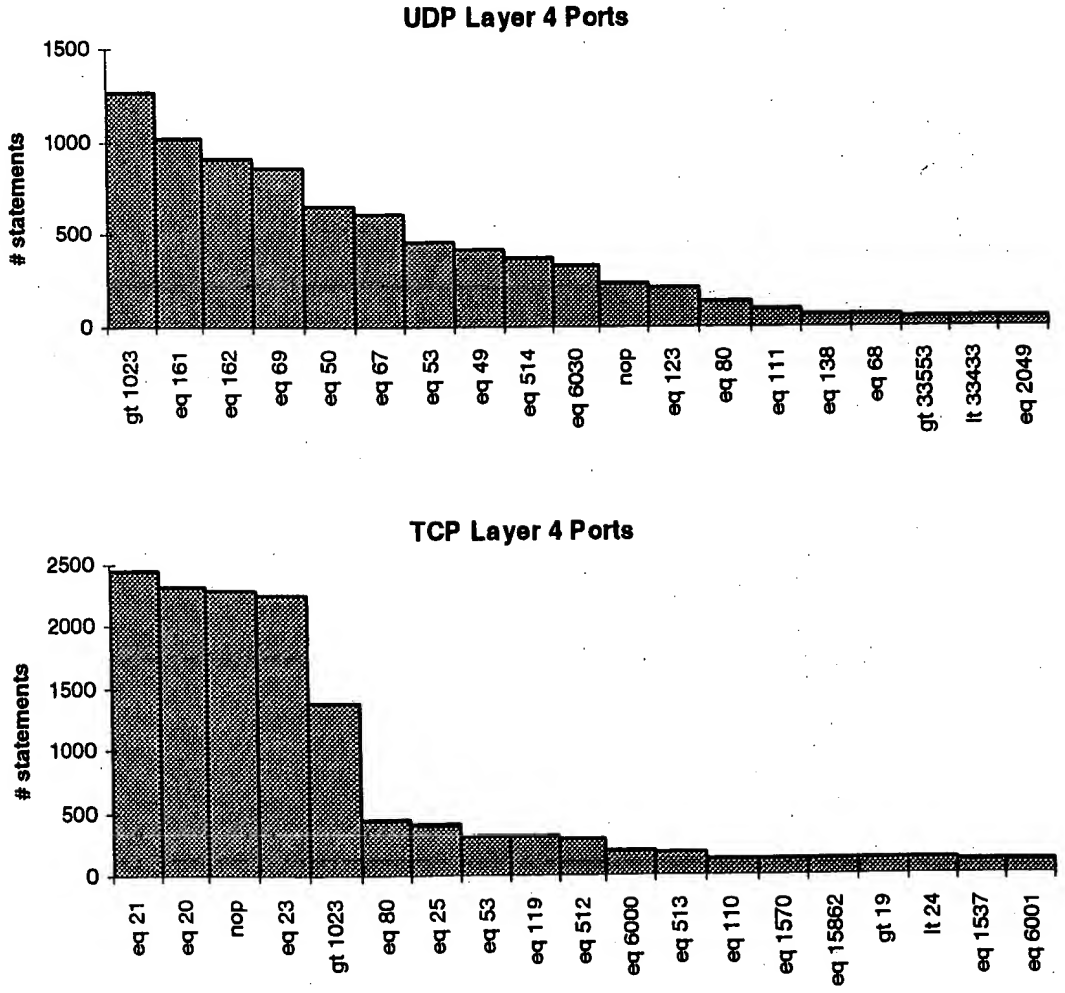

Figure-7 ACL Statistic of Layer 4 Ports

2.3.4 Non-Contiguous Masks

There are router configuration files which have ACLs with non-contiguous masks:

```
access-list 101 udp 194.155.0.251 0.0.255.0 194.72.6.64 0.0.0.15 eq snmptrap
```

About 1% of all ACL statements in the database have non-contiguous masks.

2.4 Analysis of Packet Traces

The protocol breakdown of ACL router configurations does not represent the packet traffic which flows through a router. To get somehow a more realistic picture of the packets which enter a router, we analyzed two intranet traces. We will use then the numbers to weight our results. In the Figure-8 we look at the protocol breakdown of the two recorded intranet packet traces.

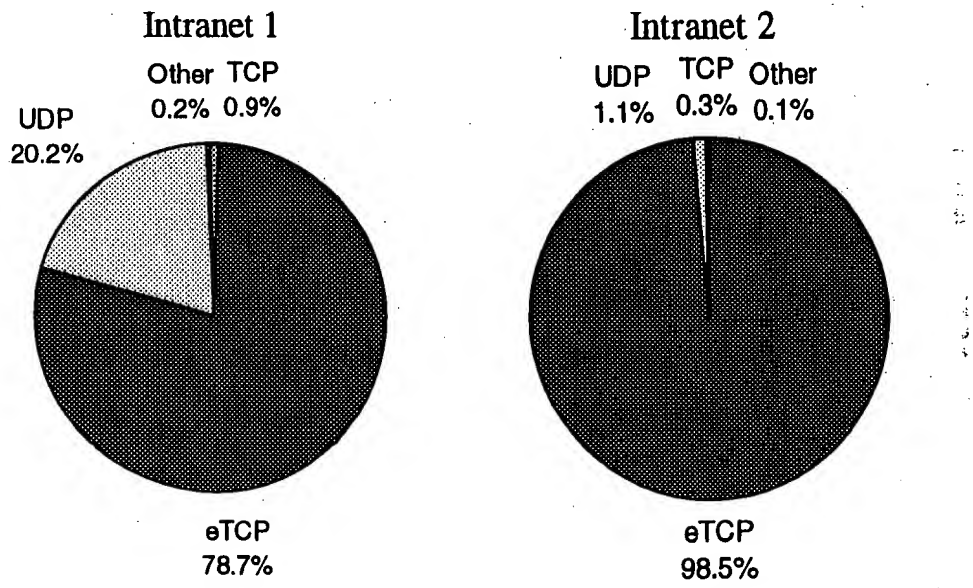

Figure-8 Packet Trace Protocol Breakdown

In the figure we see that the established TCP traffic is very dominant above the other protocols.

2.5 Summary

The analysis returned interesting numbers of the protocol breakdown. The most frequent protocol statements are TCP, UDP and IP. Other protocols are very rarely used. Sixty percent of all statements define layer 4 port addresses. The result of the packet trace analysis was that the established TCP traffic is very dominant above the other protocols.

We discovered almost in every ACL redundant statements. With preprocessing the ACLs we could delete such redundancy.

Troublesome effects are the overlapping of IP and layer 4 port addresses. Because the tree structure of the Mtrie+ compares the ACL qualifiers sequentially this can cause errors in the ACL pro-

cessing.

Another interesting fact is that in all ACL examples the source port qualifier is never used. The combination of the qualifiers established and port numbers occurred only once in all 44'000 statements. Together with the high percentage of established TCP packets in the packet traces we can see that here is a potential for an optimization.

3.0 Description of the Mtrie+

In this chapter we first describe the approach on how the Mtrie Plus Engine executes ACL processing. Then we look at the Mtrie+ data structure.

3.1 ACL Processing in the MPE

The Mtrie Plus Engine approach combines the routing with the access list processing. Both merge into an extended tree structure. We learn more about this extended tree structure in the next chapter. But first let's go through a short introduction of the MPE interfaces for the Mtrie+ and on how the MPE forwards packets to get a better picture of the environment of the ACL-Mtrie+.

3.1.1 Introduction to the MPE

The MPE is a multi-threaded "processor" that services a queue of packet headers, preprocessed into a standard packet label format. The MPE processes each packet according to the forwarding data structure Mtrie+. The Mtrie+ is an extended and modified version of the Mtrie used in IOS. When a packet arrives at an input port, the next free MPE thread starts processing it according to the assigned Mtrie+. This results in a forwarding action. The Mtrie+ includes instructions which refer to operations that demux on protocol fields, match on IP addresses etc. Let's have a closer look at the packet label.

3.1.2 Packet Label

A packet header is extended by additional information which are required to perform the operations. Specifically the Mtrie+ packet label is a 32-byte value with format:

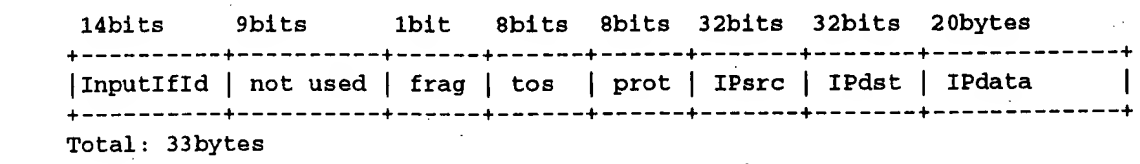

Figure-9 Packet Label

The field meanings are:

inputIfId: input interface identifier; number of the interface on which the packet arrived
frag: fragment; indicates that the packet is a part of a fragmented packet
tos: type of service
IPsrc: IP source address
IPdst: IP destination address
IPdata: the field contains the first 20 bytes if the layer 4 PDU e.g. layer 4 header. This allows to support ACLs with protocol specific rules like TCP ports or ICMP message types.

3.1.3 Status Register

The status register holds the information to which output interface a permitted packet is forwarded. This register is initialized before each lookup sequence and is modified during the lookup.

The Mtrie+ status register contains a field for the output interface ID and one for the transmit queue ID.

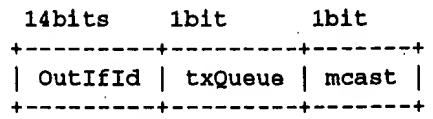

Figure-10 Status Register

The content of these registers is initialized to a given value, e.g. to the CPU port and to the non-preemptive transmit queue. If the lookup is processing a leaf node, the output interface ID and the multicast bit stored in the status register are overwritten with the appropriate values stored in the leaf.

The *txQueue* bit cannot be reset by a leaf. So for future QoS support, there can be included a leaf in the Mtrie+ structure just for increasing the priority of a given packet.

The MPE supports multicast by the *mcast* bit. If a multicast packet is processed, the *mcast* bit is set and after the completion of the Mtrie+ lookup we find in the *OutIfId* field the ID of the IP multicast forwarding bit vector.

3.1.4 Packet Forwarding

Each arriving packet label is processed through several subtrees of the Mtrie+. The input ports of the MPE hold an oppointer register which points to the next Mtrie+ subtree (Figure-11). We will give an exact definition of what an oppointer is in the following chapter. At the moment it is enough to know that it contents a pointer to another node or subtree and an opcode which indicates what operations the MPE has to execute on the packet label. The input interface ID field in the packet label indicates at which oppointer register to start with the lookup. Let us have a look at a sample lookup procedure first and afterwards at a list of all possible lookups.

A ready MPE thread starts with collecting the next packet label in the queue. The input interface ID field indicates which oppointer register to read. The oppointer register points to an ACL-Mtrie+ subtree. The opcode of the oppointer indicates the type of node at which the address is pointing. Hence the MPE will execute the right operations specified by the opcode on the designated fields in the packet label to obtain the next oppointer. The MPE repeats this operations until an oppointer points to the next subtree. The packet passes the first ACL subtree without getting dropped. The routing subtree is then the next logic subtree. Here the MPE executes the same operations as in the ACL subtree before. This results in a routing decision by reaching the leaf which points to the output ACL subtree. This output ACL subtree is assigned to an output interface. Then this subtree is processed and the packet passes this ACL subtree again without getting dropped. The last oppointer points to an output interface leaf which overwrites the status register of the MPE with the new output interface ID so that the packet gets forwarded to the right output interface. The oppointer of the output leaf points to the value *Term* which terminates the lookup. This MPE thread is free now to process the next packet label.

Here follows now a list of all possible lookups:

1. *oppointer register*: Is the starting point of a lookup. Can point to an ACL or a routing subtree. In case of a fragmented packet an additional register in the MPE is pointing directly to the routing subtree. Such packets, except the first one, get forwarded without being processed by the ACL subtrees by convention.
2. *input ACL subtree*: Can point to a routing subtree or to an output leaf. If the decision is to drop the packet then the last oppointer points to an output leaf which overwrites the status register with the drop port ID. Otherwise the routing subtree is referenced. The option exists instead of referencing the routing subtree to point to an output leaf which overwrites the status register with the cpu port ID.
3. *routing subtree*: Every branch of the routing subtree ends with an output leaf. This output leaf overwrites the status register with the output interface ID to deliver the routing decision. The *nextOP* of the output leaf can point to an ACL subtree, to an output leaf or to the value *Term*. The ACL subtree is chosen if output packet filtering is requested. If no routing decision is delivered the *nextOP* points to an output leaf which overwrites the status register with the cpu port ID. This happens if the routing subtree delivers no routing decision and the packet has to be forwarded to the cpu for further processing. Otherwise it points to *Term* to terminate the lookup.
4. *output ACL subtree*: Can point to an output leaf or to the value *Term*. If the decision is to drop the packet then the output leaf overwrites the status register with the drop port ID. Otherwise the last oppointer of the subtree points to *Term* to terminate the lookup.
5. *output leaf with cpu port ID*: Can point to an ACL subtree or to the value *Term*. If packet filtering is necessary before the packet is processed by the cpu, the *nextOP* points to an ACL subtree. Otherwise it points to *Term* to terminate the lookup.
6. *cpu ACL subtree*: Requested to inhibit cpu overload by processing unnecessary packets. Can point to an output leaf or to the value *Term*. If the decision is to drop the packet then the output leaf overwrites the status register with the drop port ID. Otherwise the oppointer points to the value *Term* to process the packet by the cpu.

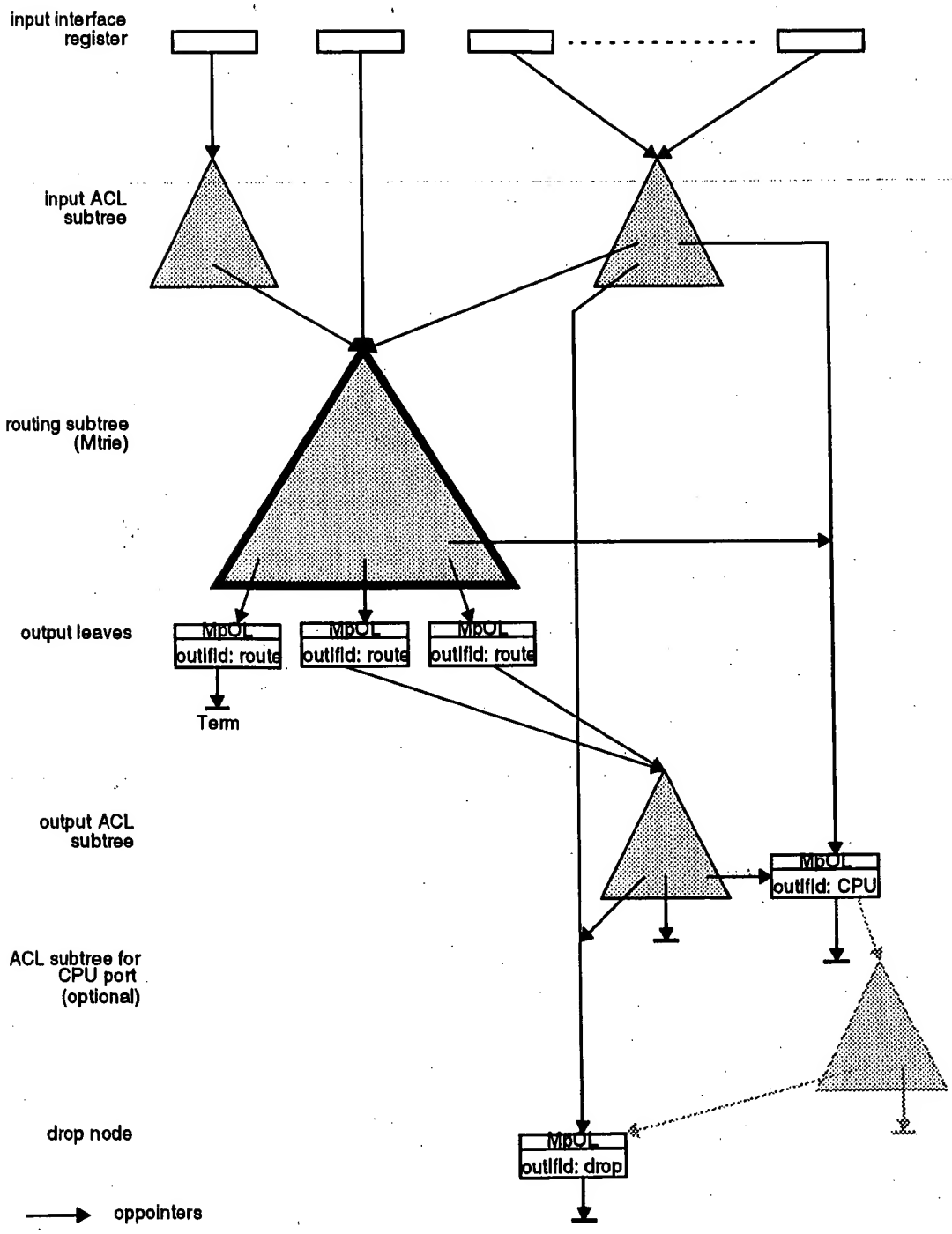

Figure-11 Mtrie+ Subtrees

3.2 Mtrie+ Data Structure

The Mtrie+ is an extension of the already known Mtrie. The current Mtrie distinguishes only between the leaf and the node type elements and is used only for routing. The Mtrie+ extends this idea and uses different node types in the tree for matching and branching on different header

fields. This allows to combine ACL processing with the routing and the multicast forwarding.

The Mtrie+ consists therefore of three different subtrees, each representing either an input ACL, an output ACL or the routing table. The multicast lookup is integrated in the routing subtree. But the routing subtree is not discussed within this project. Currently the routing subtree is the same Mtrie data structure which is used in the existing routers to generate routing decisions. That is possible because the Mtrie is a subset of the Mtrie+ data structure. The subtrees are created by translating their corresponding configuration files.

The Mtrie+ is created in the MPE off-chip SRAM by software. An MPE thread processes the header according to the Mtrie+ associated with the input port on which the packet arrived. It transverses the data structure to determine the forwarding action associated with this packet.

3.2.1 Oppointer

An Mtrie+ is a tree data structure of nodes, each specified by an oppointer that indicates the address of the node and its type, specified by an opcode. The name oppointer is a combination of the words operation and pointer. The operation is specified by the opcode and the pointer by the address. Specifically the Mtrie+ oppointer is a 32-bit value with format:

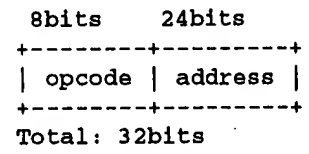

Figure-12 Oppointer

The Mtrie uses a LSB-bit hack to distinguish between nodes and leaves, e.g. the LSB-bit set indicates a node. The lookup algorithm has to be aware of that and to take the necessary operations to retrieve the address. That LSB-bit hack worked for the Mtrie but not for the Mtrie+ because of the many new node types. Therefore the 32-bit oppointer reserves 8-bit for the opcode and the other 24-bit for the address.

The address addresses on 16-byte boundaries in the MPE SRAM, the unit read by the MPE. Therefore the 24-bit address has to be shifted 4-bits before using. So a valid node address has always the lower four bits set to zero.

During the lookup the opcode of a node is indicating what operations the MPE has to execute on the packet label. The result of the operation selects the next oppointer whose address points to the next node in the tree data structure. The lookup is stopped if an oppointer refers to the specific value *Term*.

3.2.2 Node Types

Opcode	Shortcut	Type	Description
0 (0b00000000)	MpTerm	special	Mtrie+ lookup terminator. Used in the TERM value (TERM = 0x00000000 e.g. TERM = NULL)
1 ... 7	--		Reserved
8 ... 15 (0b00001xxx)	MpMN0..7	match node	Match nodes: Contains a 32-bit value and a 32-bit mask. This nodes match on 8 different 4 byte sized fields of the packet label
16 ... 31	--		Reserved
32 ... 63 (0b001xxxxx)	MpDM0..31	demux node	Demultiplexing nodes: These nodes demultiplex into different Mtrie+ branches based on different packet label byte positions. The five LSB bits specify the byte position within the packet label that should be used for demultiplexing.
64, 65 (0b0100000x)	MpMNL4src/dst	match node	Match node on the layer 4 ports
66 (0b01000010)	MpDMprotR	demux node	Reduced demux on protocol type
67 (0b01000011)	MpDMIPdstR	demux node	Demux on a prefix of the IP destination address
68 ... 71 (0b010001xx)	MpHNIPsrc0..3	hash node	Hash nodes: These nodes hash into different Mtrie+ branches based on different packet label byte positions. The two LSB bits specify the byte position within the packet label that should be used for hashing.
72 ..75 (0b010010xx)	MpHNIPdst0..3	hash node	
76, 77 (0b0100110x)	MpHNL4src/dst	hash node	Hash nodes on the layer 4 ports
78 (0b01001110)	MpMNbIP	match node	Match node on both IP address bytes
79 (0b01001111)	MpMNbIPL4	match node	Match node on both IP address bytes and layer 4 ports
80, 81 (0b0101000x)	MpDML4src/dst	demux node	Demux node on the layer 4 ports
82 ... 126	--		Reserved
127 (0b01111111)	MpOL	special	Output interface leaf
128 ... 255	--		Reserved

Table-1 Mtrie+ Node Types Overview

All node operations perform actions on one or more bytes in the Mtrie+ packet label except the leaf nodes. The result of the operation is then used for indexing the next oppointer. That oppointer points then to the next node and describes the type of it. There are four different main node types:

1. *Demultiplexing nodes*: These nodes demultiplex into different Mtrie+ branches based on different packet label byte positions.
2. *Matching nodes*: These nodes compare given byte positions of the packet label to given node data. The result indexes the next oppointer.
3. *Hashing nodes*: These nodes hash into different Mtrie+ branches based on different packet label byte positions.
4. *Specialized nodes*: They terminate the lookup process or perform operations which can not be done by the other nodes.

3.2.2.1 MpDM - Generic Demux Node

This node demultiplexes the different packet label byte-fields. Each node contains an array of 256 oppointers, one for each possible value of the according packet label field.

The 5 LSBs of the opcode determine which of the packet label bytes should be used as an index for the oppointer array within the node.

So a single Mtrie+ lookup step consists of:

1. Reading the packet label byte position encoded in the LSBs of the opcode
2. Reading the demux node array entry identified by the read packet label value
3. Executing the oppointer read from the array entry

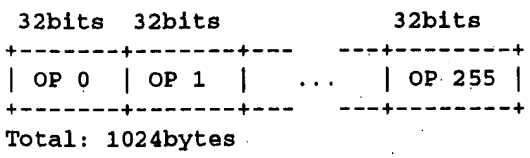

Figure-13 Demux Node

3.2.2.2 MpDMprotR - Reduced Protocol Demux Node

As the most frequent protocol types in ACL entries are IP, UDP, TCP and established TCP a protocol demux node with a reduced demultiplexing facility is provided.

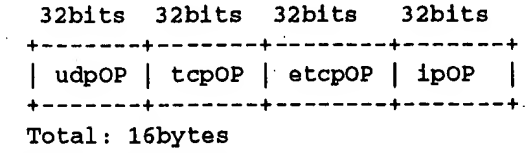

Figure-14 Reduced Protocol Demux Node

Compared to the MpDM node this node requires only 16 bytes of memory instead of 1024 bytes.

Supporting established TCP as a protocol type within this node has the advantage that we don't have a two level lookup for identifying established TCP. Otherwise we must first identify TCP with a demux or match node and then in a second step we use a match node to distinguish between an already established connection and packets initializing a connection by testing the SYN bit field.

3.2.2.3 MpMN - Match Node

The match node compares the 4 byte packet label-word (identified by the LSBs of the opcode) to the pattern field stored in the match node. The mask field determines which bits are compared. In case of a match the oppointer *matchOP* is executed otherwise *noMatchOP* is executed.

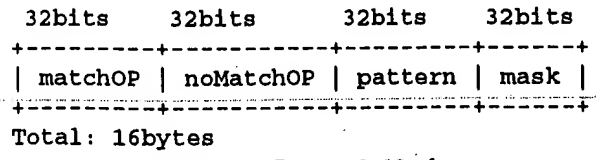

Figure-15 Match Node

3.2.2.4 MpMNL4 - L4 Port Match Node

This node contains two 16-bit values for specifying a layer 4 port number range and two oppointers. Similar to the preceding node, one oppointer is used in the matching case and the other one in the non-matching case.

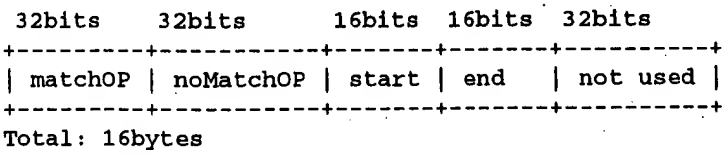

Figure-16 Match Layer 4 Address Node

3.2.2.5 MpHN - Hash Node

This node hashes the different packet label byte-fields (identified by the LSBs of the opcode). Each node contains 12 buckets. The number of buckets can be changed. The correct bucket is selected with the division method.

The LSBs of the opcode determine which of the packet label bytes should be used as an index for the oppointer array within the node.

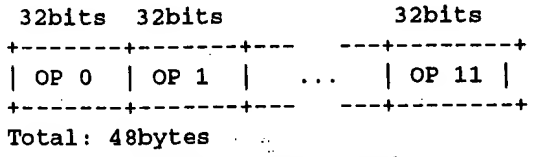

Figure-17 Hash Node

3.2.2.6 MpHNL4 - L4 Hash Node

Here this node hashes the 2byte port numbers to its oppointer array. The index is calculated by selecting the correct bytes in the packet label determined by the LSBs of the opcode. The array of oppointers look like in figure 17.

3.2.2.7 MpDMIPdstR - IP Destination Address Demux Root Node

This is a demultiplexing node that contains 2'113'680 oppointers. This node allows it to demultiplex the 24bit prefix part of all the IP destination addresses using just one memory read operation.

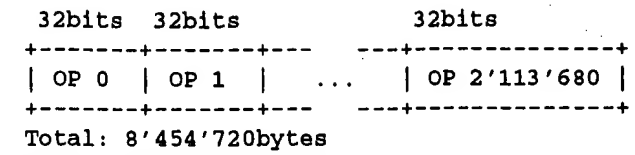

Figure-18 IP Destination Address Demux Root Node

This node is conceived to be used as the root node of the routing Mtrie+. So it is used only once within the whole Mtrie+ structure.

3.2.2.8 MpMNbIP - Source and Destination IP Address Match Node

This node compares both 4byte packet label-words to the pattern fields stored in this match node. The mask field determines which bits are compared. In case of a match the oppointer *matchOP* is executed otherwise *noMatchOP* is executed.

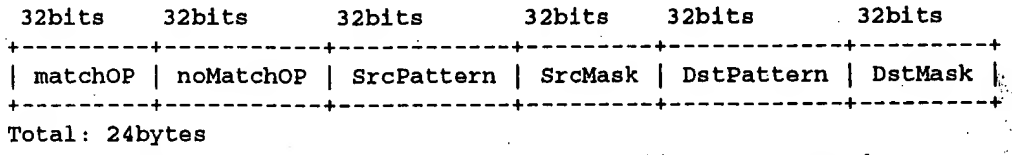

Figure-19 Source and Destination IP Address Match Node

3.2.2.9 MpMNbIPL4 - Src/Dst IP Address and Layer 4 Port Match Node

Like in the MpMNbIP node here additionally the layer 4 ports are compared.

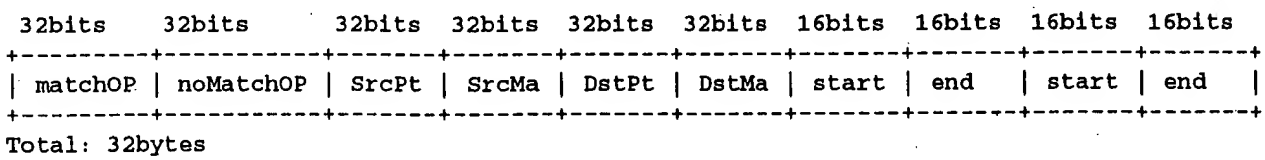

Figure-20 Source and Destination IP Address and Layer 4 Port Match Node

3.2.2.10 MpDML4 - Layer 4 Port Demux Node

Because about 60% of all ACL statements use the layer 4 port qualifiers, this node is demuxing on them. This node is especially large because we have to demux on a 2byte packet label-field. Therefore they should be used close to the root node.

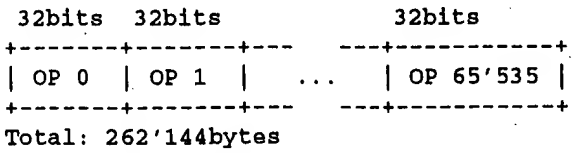

Figure-21 Layer 4 Port Node

3.2.2.11 MpOL - Output Interface Leaf

The status register is overwritten with the values provided in this node.

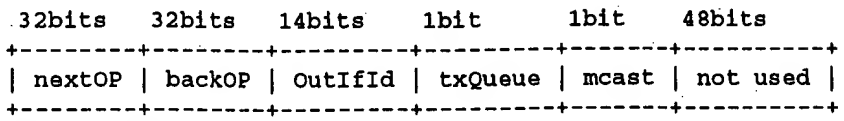

Figure-22 Output Interface Leaf

This Mtrie+ node type typically terminates a Mtrie+ lookup (*nextOP* is set to *Term*). But it also can be used just for changing the status register contents. The oppointer provided in the *nextOP* field refers to the next Mtrie+ node in this case.

The *backOP* and the *mcast* is used in the routing subtree. If an existing route is overwritten by a more specific one, the new MpOL node is inserted before the old one. This results in a linked list of output interface leafs. So the old route is recovered when the more specific route is removed. The *mcast* bit indicates that the MPE has to process a multicast with the current packet.

In case the fragment bit within the Mtrie+ packet label is set, the MPE terminates the lookup in any case regardless the *nextOP* value. This guarantees the standard behavior of a router, which does not check fragments on any ACLs.

There is one default drop node. It is a MpOL node with the *OutIfId* set to the drop port ID. The oppointers of new created nodes are pointing to this global default drop node. This has two meanings: In the building process of the data structure it tells the algorithm that this oppointer is unused and is free for pointing to an another node. In the lookup process it generates a drop packet decision.

3.2.2.12 MpTerm - Termination Leaf

If an oppointer points to this leaf node, the lookup is terminated. We need only one default termination leaf node in the whole data structure.

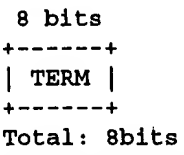

Figure-23 Termination Leaf

3.3 Summary

Merging the routing and ACL processing into one data structure is the approach of the MPE. The MPE is a multi-threaded "processor" which uses the Mtrie+ data structure to forward packets.

In the section "3.2 Mtrie+ Data Structure" we learned that the basic building block of all nodes is the oppointer. It contains the address of the next node and its opcode. The opcode describes what action the MPE has to do on the packet label in order to select the right next oppointer. We intro-

duced all different nodetypes and described what operations the MPE has to execute on the packet label to find the next oppointer. The ACL-Mtrie+ is a subtree of the whole Mtrie+ data structure of the MPE. The other subtree is the routing subtree and is not discussed in this project.

4.0 Solution based on ACL-Mtrie+

In this chapter we illustrate how to compile an ACL-textfile to an ACL-Mtrie+ in the simulation. First we specify the properties of the simulation environment, followed by the description of the implementation goals and considerations. Then we present the actual implementation and how an ACL-Mtrie+ is built.

4.1 Specification of the Simulation

The simulation is written in C++. There are no alternatives to explore the Mtrie+ approach and there was already a lot of code available for processing ACLs.

The router configuration files are textfiles and contains all information we need. A parser algorithm translates them into a router object so that we collect only the data we need and have easy access to it. During this operation we can get rid of redundant ACL statements.

The ACL-Mtrie+ constructing algorithm has to fulfill several requirements. First we need a flexible architecture. So we are able to do changes quickly or to extend the algorithm without great effort. Second it has to keep track of several parameters. Those are how many nodes and what type they are, and record the depth of internal node structures like linked lists. Then we are able to determine how much memory the tree is using.

The lookup algorithm keeps track of how many steps in the minimum, maximum and average have been executed in the ACL-Mtrie+. So we can see how effective the latest change was.

To check if the forward decision we receive from the ACL-Mtrie+ is correct, we need a verification. A simple packet filter tests the generated test packet label against each statement in the ACL sequentially. Then its output is compared with the one of the ACL-Mtrie+.

The test packet label are generated by the test algorithm. Here for each ACL statement a set of test packet labels are generated, then fed into the ACL-Mtrie+ and the simple packet filter, and then compared. If the forward decisions are not equal, a error message indicates what ACL statement produced the mismatch. This will help to track down wrong tree structure.

4.2 Implementation Goals and Consideration

The goal of the ACL-Mtrie+ simulation is to keep the lookup count and the memory usage low. The forwarding rate of the MPE is related to the number of lookup steps in the Mtrie+. Hence it is critical to keep that parameter to a minimum in order to maintain a high forwarding rate.

The ACL is interpreted sequentially from the beginning on e.g. we use the first ACL statement for constructing the first branch in the ACL-Mtrie+, then we advance to the second ACL statement and we add the necessary nodes and references etc.

For every protocol we define a different sequence of nodes. Mainly we distinguish between protocols using portnumbers (as UDP and TCP) and all the other protocols. But also for instance ICMP or IGMP may have their special sequence of construction for supporting ACL statements depending on ICMP / IGMP message types etc.

4.3 Implementation of the ACL-Mtrie+

In this section we present the implementation of the ACL-Mtrie+. In the first approach of this project we tried to compile the ACLs into a traditional tree data structure. We will discuss why it can't work. A new approach masters this problem and delivers correct forwarding decisions.

4.3.1 Problem with the Standard Tree Structure

In the first approach we tried to compile an ACL to a standard tree structure. The standard tree structure consists of tree different kind of nodes: A root node, branch nodes and leaf nodes. The root and branch nodes have several links. Each link is pointing to different branch or leaf nodes, e.g. a branch node could point to 256 different branch nodes. We add all the necessary nodes to the tree statement by statement. If we want to preserve the order of the ACL statements, we have to demux on all qualifiers of a statement, i.e. we need to demux on the protocol, on the layer 4 port numbers, and on all single IP address bytes. Otherwise it could happen, that statements are overlapping with other statements thus violating the semantic of the ACL. As we can imagine this leads to a big memory usage considering that every demux node is of the size of one kilobyte. The order of growth for the memory is exponentially and therefore this approach can not be realized.

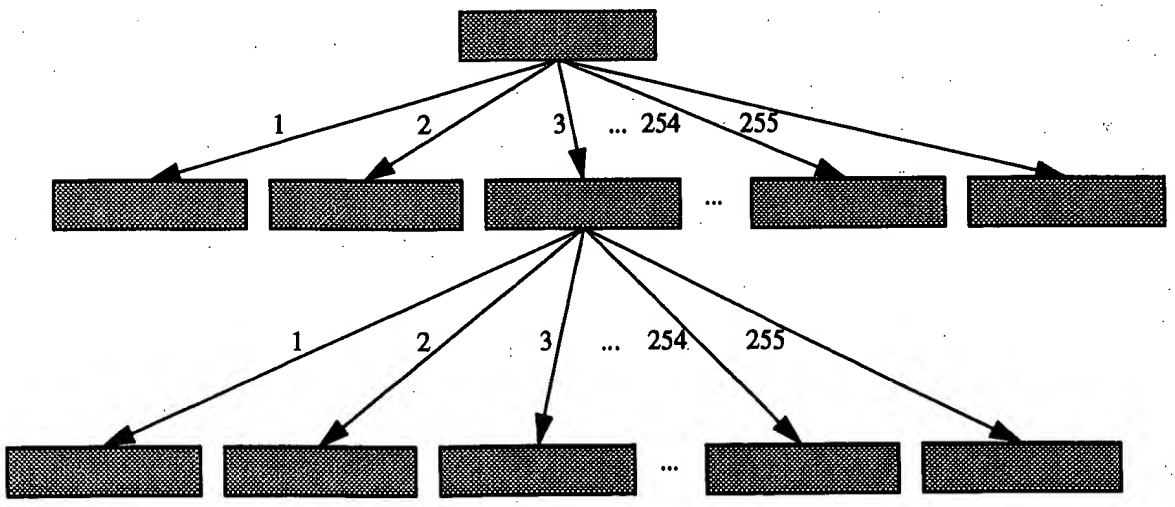

Figure-24 Traditional Tree Structure

Of course if we analyze an ACL thourougly we discover that not all demux nodes are necessary because some qualifiers in the ACL have the same value. Consequently it makes no difference, if we demux on this qualifier or not. But then we can not expect that one individual sequence of demux nodes delivers correct forwarding filtering decisions for other ACLs. Therefore we could analyse every single ACL to extract the correct sequence of demux nodes before we start to build the tree data structure. But even with only a few demux nodes the amount of necessary memory is big with this approach. That's because in every ACL we find statements with 'any' qualifiers. This means that then all 256 links are pointing to other demux nodes (Figure-24). With three demux nodes we need already more than 65 megabytes of memory space and usually three demux nodes are not sufficient enough.

4.3.2 New Correct Data Structure

We need to save memory space and lookup steps. How can we achieve this without violating the ACL semantic?

4.3.2.1 Linked List of Match Nodes

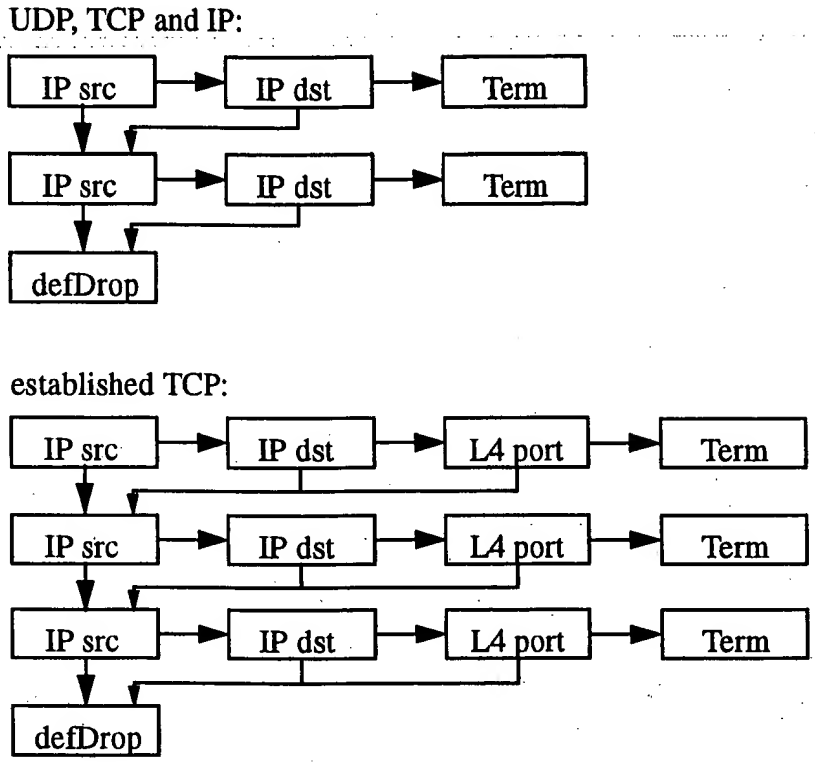

Figure-25 Linked List of Match Nodes

A simple approach in order to prevent the ACL semantic is to compile every statement in a linked list of match nodes (LM). Then every *noMatch* link points to the next LM which represents the next ACL statement. We go on with this procedure until we have processed all statements. But then this data structure is not very different from the already existing implementation of ACLs in IOS. This data structure does not need a lot of memory space, but the lookup step count is unacceptable high.

4.3.2.2 Tree of Demux Nodes for the IP Address

We need to lower the depth of the linked lists for decreasing the lookup count. Therefore we demux on the IP address and insert the appropriate demux nodes (Figure-26). This leads then to a tree before the linked lists which subdivides the list in many smaller ones. Like before, if we look closer at the ACL statements we discover that there is a potential for optimizing the tree structure for decreasing the memory space. Particularly we can spare some demux nodes because some bytes of the IP masks have the value 255. An example could help:

```
access-list 142 permit ip any 152.163.0.0 0.0.255.255
access-list 142 deny ip any any
```

Here we don't need to demux on the source IP address bytes because in both lines they have the same value. Only the destination IP address is different. So we directly point to the demux node of the first byte of the destination IP address. We saved all demux nodes for the source IP address.

We also don't need a full demux for the destination IP address. The first line generates two demux nodes because the links are exactly specified. The second line is less specific than the first line. Therefore we don't insert additional demux nodes. All unused links of the more specific demux node point to the second line. The used links point already to the first line. So we append the second line to it. This is necessary to preserve the ACL semantic.

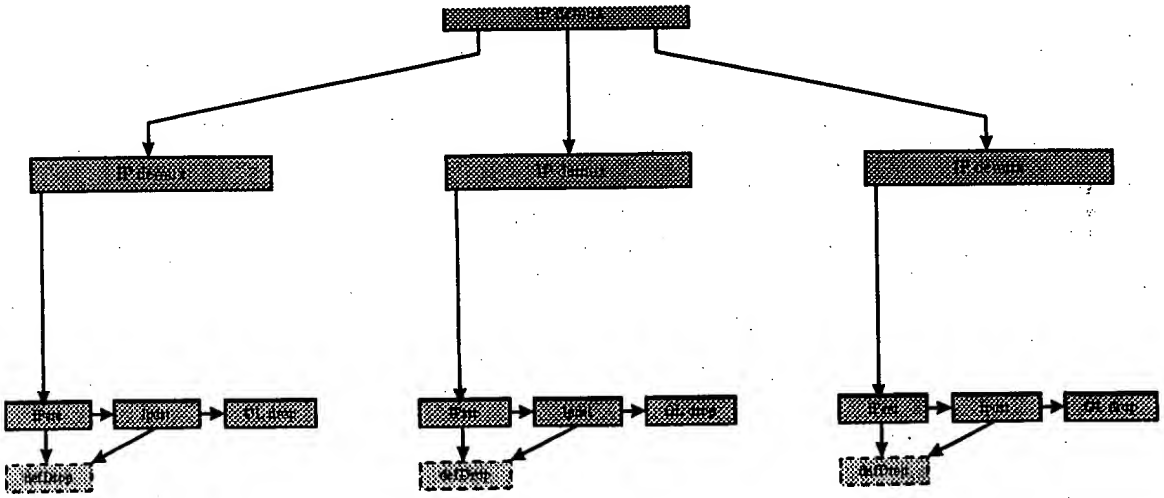

Figure-26 IP Address Demux Nodes

4.3.2.3 Demux Nodes for the Layer 4 Port Numbers

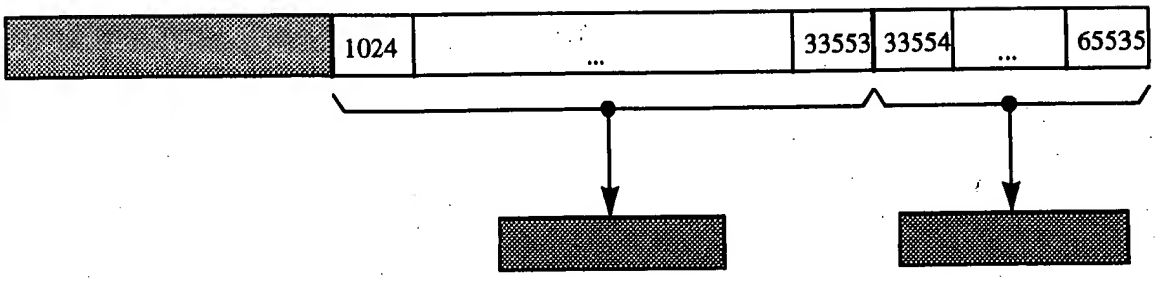

Figure-27 Layer 4 Port Demux Node

We know from the analysis of ACLs that a lot of statements which use the layer 4 port numbers are organized in groups. Their IP addresses are the same and they can only be distinguished by their layer 4 port numbers. So we can further decrease the depth of the lists if we first demux on

the layer 4 port numbers (Figure-27). This is complicated by the fact that the ranges of the port numbers can overlap. We need to split up the ranges into sectors, e.g. *gt 1023* and *gt 33553* result in sectors 1024-33553 and 33554-65535. This gives us the advantage that all links in one sector point to the same next node. This saves us a lot of memory space. The consequence is that now the processing of one statement is more complex. The reason for it is that we have to process the statement with the qualifier *gt 1023* in both sectors. We do that by stepping through all possible port numbers and then process this statement for every valid sector.

4.3.2.4 Reduced Demux Node for the Protocol

The last optimization for cutting down the depth of the list is inserting at the beginning a reduced demux node for the protocol. We have separate links for UDP, TCP, established TCP and IP. Protocols like ICMP and IGRP and many others follow the IP link. To distinguish also among them we let the IP link point to a full protocol demux node.

A further advantage is the link for the established TCP statements. The analysis of the ACL showed us that we have usually very few statements for established TCP packets. Of all ACLs with established TCP statements 51% have only one statement which permits all traffic for it. For those ACLs the link of the reduced demux node for the established TCP points directly to a leaf node. That results in a very low lookup count for established TCP packets.

4.3.3 Building an ACL-Mtrie+

Here we illustrate how the simulation builds the ACL-Mtrie+. We start with parsing the router configuration file into a router object. Next we cycle through all access lists in this router object. For each access list we create a manager object. It analyzes first the ACL and the result is selecting the appropriate node sequences. After creating the root node all created nodes according to the statements in the ACL are appended to the tree. Before the final step we verify the filter decisions by generating many test packet labels for each ACL statement and feed them into the ACL-Mtrie+ and a simple filter algorithm. Both filter decisions are then compared. Finally we print all the gathered statistical information before we advance to the next ACL.

4.3.3.1 Parsing the Router Configuration File

The ACL parser reads the router configuration file line by line and appends the parsed line to the access list of the router object. It interprets only the lines that are useful for this simulation.

As we know from the ACL analysis there are redundant statements. Before appending a new line to the access list of the router object we try to find a match in the already parsed ones. If there is a match, we don't add it to the list because it is redundant.

4.3.3.2 Node Sequence List

The node sequence list describes in what sequence the nodes are created for each statement and added to the ACL-Mtrie+. The node sequence list is different for certain statements. This allows for a more dynamic node structure in the ACL-Mtrie+. An example could help:

for UDP:	for established TCP:
demux_protR	demux_protR

demux_l4_dst	leaf
demux_ip_src0	stop
demux_ip_src1	
demux_ip_src2	
demux_ip_src3	
match_list	
stop	

Here both examples start with the reduced protocol node. But because in this example ACL we have only one established TCP statement which allows all its traffic, we point directly to a leaf node. This is saving us unnecessary nodes and therefore we reduce the lookup count and memory usage. In the UDP example we use additional demux nodes to subdivide the LM in smaller pieces. The expression *match_list* stands for the LM.

The node sequence lists are giving us now a great flexibility in what kind of nodes to use before the LM or even neglect the LM and replace it with a leaf node.

4.3.3.3 Adapting to Individual ACL

Because the builder algorithm is using different node sequences for different statements within an ACL, the node structure of the ACL-Mtrie+ is more adaptive in respect to the individual ACL.

First before building the ACL-Mtrie+, we analyze the ACL. With the results of the analysis and with heuristic rules we select the appropriate node sequences for this ACL.

We know that the first node is always a reduced protocol node. This gives us the advantage to directly process established TCP statements. Therefore it is logical to distinguish mainly between the four different branches UDP, TCP, established TCP and IP. So each of those four branches has its own node sequence list.

The UDP and TCP branch look basically very similar but they can differ depending on the ACL analysis. Both use layer 4 port numbers and that's the reason why they use always a demux layer 4 destination node after the reduced demux protocol node. When creating a demux layer 4 node it is always partitioned into sectors and then each sector is initialized. Then all links of each sector point to a demux IP node node. Therefore the first three nodes are always a reduced demux protocol, a demux layer 4 and a demux IP node. Usually now the next nodes are additional demux IP nodes and then the last entry in the node sequence list is the designator *match_list* for the LM. But if the number of statements is less than seven, it's unnecessary to use all those demux nodes. They only add up lookup steps and memory but do not help subdividing the list. Therefore the node sequence list for this case does not use any additional demux nodes but the first three one.

For the established TCP branch we have one additional case. If there is only one statement which allows all established TCP traffic, we don't need any demux nodes and a LM after the reduced demux protocol node. Therefore the link of the demux_protR is pointing directly to a leaf node.

For the IP branch we have similar cases like for the UDP/TCP branch, except that it does not use layer 4 port numbers. Instead it is adding a full demux protocol node after the demux_protR node. That's necessary because in this branch we add all the other protocols than UDP and TCP. So we need a full demux protocol node to distinguish among them. Therefore the first two nodes for this

branch are always a reduced demux protocol and a full demux protocol node. Then additional demux nodes are following if they help reducing the depth of the LM.

The complete list of node sequences and the conditions when they are used are illustrated in the 'Appendix D: Sequence List Configurations'.

4.3.3.4 Process ACL Statements

Each statement is now creating a node according to the picked node sequence list and the current node type. The building algorithm then is examining the oppointer links of the current node, which are selected by the node type of the current node and the qualifiers of the statement. If they are unused, they are pointing now to the new node, appending it to the data structure. In both cases, used or unused links, the builder algorithm recursively follows this link down and continues there but in respect to this node type.

The node sequence list describe always the maximum number of nodes for each statement. But not all IP demux nodes are necessary. If the byte of the mask determined by the node type has the value 255, then it is redundant and can be neglected. That is because all links of this node would point to a single other node. This is resulting then in just a useless additional lookup step. Therefore this node is not added to the ACL-Mtrie+ and skipped.

It is not important in what sequence the nodes are appearing in the node sequence list. For instance a possible sequence could be: demux_ip_src0, demux_ip_dst3, demux_ip_src2, etc. This opens a great flexibility and opportunity in optimizing the node structure individually to an ACL.

Finally we arrive at the linked list of match nodes. The list has several lines of match nodes. Each lines represents an ACL statement. As we know a match node has two links. The match link points to the next match or leaf node. The noMatch link points to the next line. The Figure-25 shows the structure of the list with two different statements. When we append a new statement to the list, we check first for a match. If there is a match, we do not add the statement to the end of this list.

In the Figure-25 a single line in LM is consisting of two different match nodes. Even if the first line in the LM is matching, the lookup algorithm is going through both match nodes, resulting in two lookup steps. Because within a LM the sequence of the match nodes never changes, we can merge them into one match node, thus resulting in one lookup step. But the merged match nodes use more memory space. In order to still read them in one memory access, this has to be considered in the design for the Mtrie Plus Engine (MPE). But the advantage is that it saves lookup steps and a little bit of memory space.

access-list 101 permit tcp any host 12.34.56.10 eq 21
access-list 101 permit tcp any host 12.34.78.90 eq 21

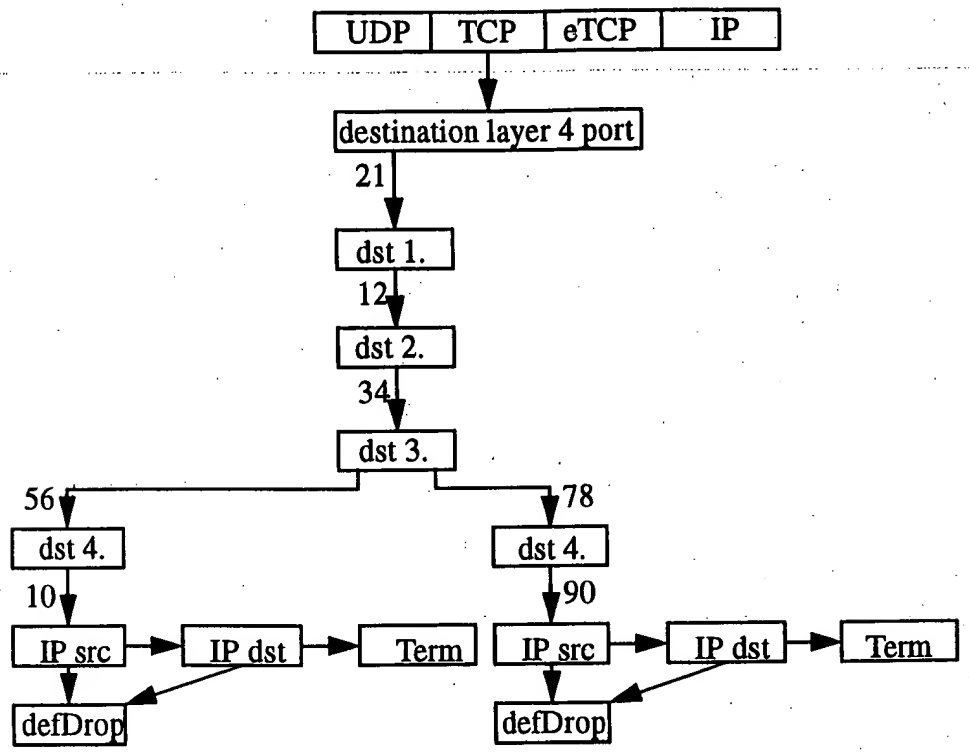

Figure-28 Example Mtrie+ Data Structure

In the Appendix "Example in how Subdivide the Lists" on page 51 the process of decreasing the depth of the lists by inserting additional nodes is illustrated with an example ACL.

4.3.3.5 Verification

We have to verify if the output of the ACL-Mtrie+ is correct. For this reason we generate for each ACL statement several packet labels so that they match this statement. Then we feed the test packet labels into the ACL-Mtrie+ and a simple filtering object and compare both filter decisions. This filtering object is doing the lookup in a similar way like the IOS software. Its simple implementation makes it easy to check the correctness of the filtering decisions.

4.3.3.6 Statistical Information

During the whole simulation we gather several informations. During the building of the data structure we count the numbers of the used nodes and the depth of the lists. During the lookup with the test packet labels we count the number of steps for each protocol. We also memorize the minimum and the maximum lookup count.

With this information we can calculate the total amount of used memory space for this data structure and the average lookup counts for each protocol. The lookup count has a direct impact on the packet forwarding rate of the MPE. The used memory space gives a picture of how big the memory requirements in a router for the ACL-Mtrie+ are.

4.4 Summary

In the section "4.1 Specification of the Simulation" we specified the requirements of the simulation. A flexible software architecture supports the changes or extensions to the code. With a verification operation we check the correctness of the data structure. For gathering statistical information we keep track of internal data of the ACL-Mtrie+.

The goal of the ACL-Mtrie+ simulation is to minimize the lookup count and the used memory space. The ACL statements are processed sequentially. They are added to the tree line by line. Each protocol has different qualifiers. Therefore the sequence of the nodes, i.e. how we add the nodes to the tree, is different according to the statement and protocols too.

In the section "4.3 Implementation of the ACL-Mtrie+" we illustrated how the simulation builds an ACL-Mtrie+ with the new approach. The node sequences are providing a big flexibility for adapting and optimizing the data structure to individual ACL. Merging the match nodes into one match node reduces the lookup count further.

5.0 Evaluation

Here we evaluate different ACL-Mtrie+ in terms of lookup steps and memory usage. We compare them with the other approaches like the current software ACL processing and the soon available CAM. Next we discuss where the problems are in this implementation. Finally we discuss how the ACL-Mtrie+ could be improved.

5.1 Evaluation of the ACL-Mtrie+

The evaluation is based on the statistical numbers we gathered throughout our simulation. Specifically we recorded the mean lookup count of each protocol UDP, TCP, established TCP, IP and over all protocols. Then the minimal and maximal lookup count. Further the depth of the linked lists of match nodes. Finally the number of all used nodes and their memory usage separated by their type.

The over 130 ACLs in the database includes customers of all kind. There are ACLs from Internet Service Providers, Banks, Insurance Companies, Automobile Manufacturer, Express Shipment Companies, Research Labs etc. The results of the evaluation are across all those customers and therefore it gives only a general picture.

We distinguish between the expected and adversary case. The expected case occurs if our implemented optimizations are effective, i.e. we have a lot of established TCP packets in the traffic pattern. We simulate such a packet traffic by weighting the lookup counts according to the numbers we have from our packet traffic analysis. The weights are 1% for UDP, 0.8% for TCP, 98% for established TCP and 0.2% for IP protocols. The adversary case scenario occurs when an adversary tries deliberately to slow down the router. It does that by sending packets to the router which hit the non-optimized part in the ACL-Mtrie+ resulting in big lookup counts. The adversary case occurs if our optimizations are not effective, i.e. the packet traffic pattern is random. We simulate this by weighting all lookup counts equally, i.e. the weights are 25% for all protocols. But this also assumes that all ports receive such adversary packets which is very unlikely. Usually only a few ports of the router are connected to untrusted links such as border routers.

Nr.	#entries	UDP	TCP	ETCP	IP	min	max	expected case	adversary case
1	190	4	8.1	1	5.6	1	10	1.1	4.7
2	319	9	5	2.5	4.3	2	11	2.6	5.3
3	3731	10.7	10.2	9.6	10.4	3	12	9.6	10.2

Table-2 Lookup Counts

Nr	expected case [mpps]	adversary case [mpps]
1	136.4	31.9
2	57.7	28.3
3	15.6	14.7

Table-3 Packet Forwarding Rate

In the Table-2 we collected all lookup counts of three selected ACLs. The mean lookup count for the expected cases are so low, because of the high percentage of established TCP packets. Hence it has a great influence on the weighted mean lookup count.

The MPE has enough threads to keep the memory busy all the time. Therefore if the Mtrie+ memory runs with 150 MHz we have 150 million memory accesses per second. Because of the wide memory bus of the MPE a node is read in one clock cycle. Hence the forwarding rate for the best case of only one lookup count is then 150 mpps. The Table-3 shows the forwarding rate for the expected case and adversary case for the three selected ACLs.

The optimization for the established TCP is very effective in the expected case 1. Here the right node sequence is reducing the lookup count to one step. It leads to a packet forwarding rate of 136.4 mpps. As soon as the optimization is no more effective, the packet forwarding rate drops down to about 31.9 mpps. The result of the total mean lookup count over all 130 ACLs in the database was about 5.5 lookup counts in the best case (27.3 mpps) and 6.5 lookup counts in the adversary case (23.1 mpps). The Figure-29 and Figure-30 show the distribution of the lookup counts for the expected and the adversary case across all ACLs.

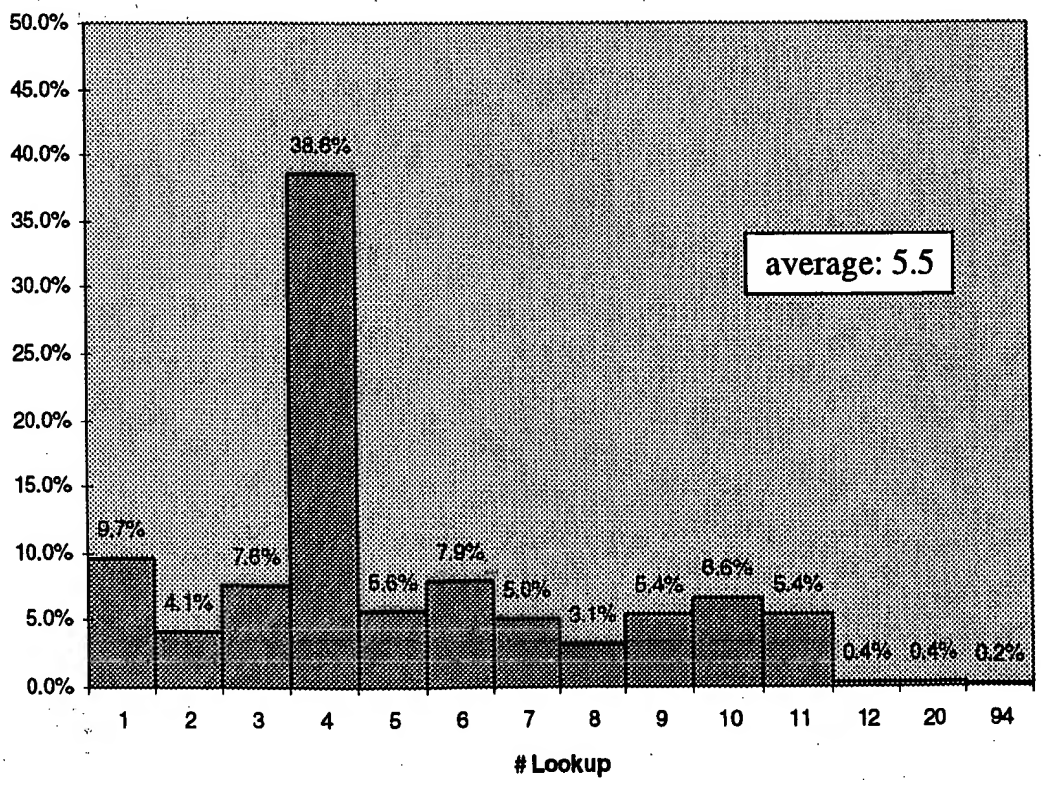

Figure-29 Lookup Count for the Optimized Case

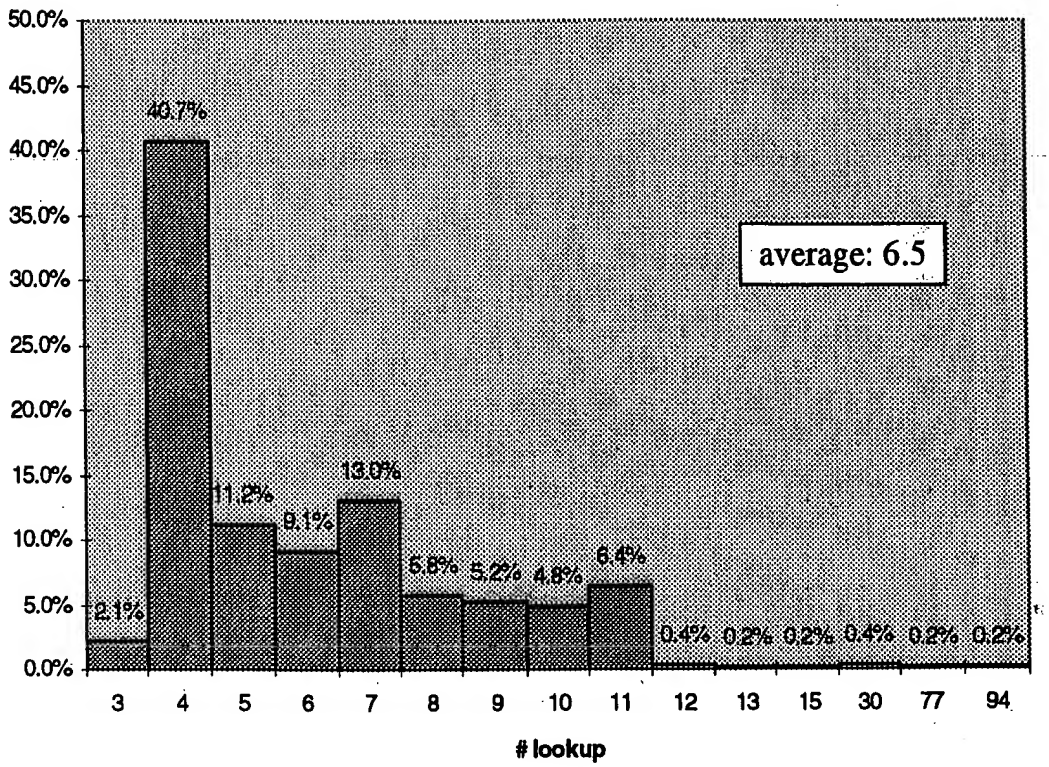

Figure-30 Lookup Count for the Adversary Case

In the Table-4 we measured the memory usage and the depth of the lists. The maximum memory requirement for a router configuration is 10.7 MBytes. But mainly the memory distribution shows that the memory usage is in the range of 0.1 to 2MBytes (Figure-31). High memory usage is not caused by a large number of statements. The reason is that there are a lot of different ACL within a router configuration, thus producing a lot of initial nodes like demux_l4_dst nodes which is using a 1/4MByte of memory for itself.

Mostly the length of the lists is only one line. There are a few ACLs which have extraordinary long lists. This problem is discussed in '5.3.1 Long List Depth'.

Nr	memory[MB]	UDP	TCP	ETCP	IP
1	0.82	1	5	0	5
2	0.66	2	1	1	3
3	2.62	2	1	1	1

Table-4 Memory Usage and List Depth

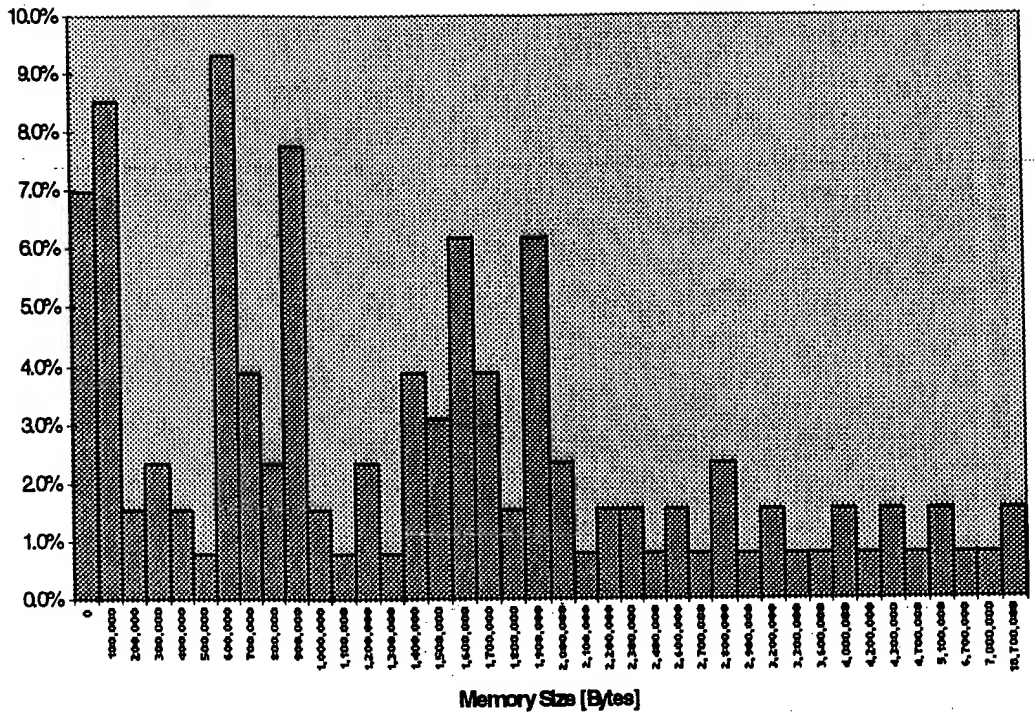

Figure-31 Memory Usage per Router Configuration

5.2 Comparison

Without the demux nodes before the linked lists of match nodes, the data structure of the ACL-Mtrie+ and of the IOS traffic filter look very similar. Both have in common that their lookup count is not constant. But each packet in the IOS traffic filter is processed by the routers cpu. This additional overhead reduces its forwarding rate. The MPE uses the several threads to keep the Mtrie+ memory busy all the time. With a wide memory bus architecture it is able to read in each memory cycle a complete node. Hence with a 150 MHz clock this results in 150 million memory accesses per second.

The advantage of the CAM in comparison to the ACL-Mtrie+ is the constant lookup count. A filter decision is always available after one clockcycle. But the MPE uses a SRAM for the Mtrie+ data structure and it has many advantages over the CAM. The SRAMs are faster, cheaper, consume less power and are larger.

Another difference is that in the CAM the routing device is separated from the filtering device. That's because the CAMs are currently not big enough to store the information of the big routing tables. Therefore a router with the CAM approach is still using a SRAM with a Mtrie data structure for generating routing decisions. In the MPE the routing and the filtering is merged into one device.

The Table-5 compares some estimated numbers between these three approaches.

The forwarding rate of the MPE can be doubled. This is achieved if the data structure is duplicated in a second Mtrie+ memory. If the MPE provides enough threads to keep both memories busy all the time, then it has 300 million memory accesses per second.

Topic	Mtrie+	IOS ACL	Cisco CAM
Clockspeed	150 MHz	--	50-66 MHz
Memory	SRAM	DRAM	CAM
Operation	each packet is checked	only the first packet of a flow is checked	each packet is checked
Lookup Count	variable optimized 1.1 expected 6	variable	constant 1
Throughput of ACL processing	optimized 136.4 mpps expected 25 mpps doubled 50 mpps	10-45 kpps	50-66 mpps
Application	ACL Routing	ACL	ACL

Table-5 Comparison between the Mtrie+, IOS and CAM approach

5.3 Problems

5.3.1 Long List Depth

In a few ACL-Mtrie+ data structures the linked lists of match nodes have a big depth. That's because the depth of the list increases if a more specific statement appends its line to a less specific one. Here is an extract of an ACL where this effect occurs:

```
access-list 121 permit udp 152.163.160.0 0.0.3.255 152.163.10.14 0.0.0.0 gt 1023
access-list 121 permit udp 152.163.161.106 0.0.0.0 152.163.0.0 0.0.255.255 gt 1023
access-list 121 permit udp 152.163.160.200 0.0.0.0 152.163.0.0 0.0.255.255 gt 1023
access-list 121 permit udp 152.163.160.201 0.0.0.0 152.163.0.0 0.0.255.255 gt 1023
access-list 121 permit udp 152.163.160.202 0.0.0.0 152.163.0.0 0.0.255.255 gt 1023
access-list 121 permit udp 152.163.160.203 0.0.0.0 152.163.0.0 0.0.255.255 gt 1023
access-list 121 permit udp 152.163.160.204 0.0.0.0 152.163.0.0 0.0.255.255 gt 1023
```

The first less specific statement advises the building algorithm to skip one demux node. Originally the following more specific statements skip the same node as well. In order to have a more rigid separation, I developed an algorithm which inserts the necessary demux nodes if a more specific statement demands it. This involved quite a complex code because know this part of the subtree must be duplicated in order to preserve the ACL semantic. Then all the links of the new inserted demux nodes are pointing then to this new subtree. But we can easily see that this produces a lot of additional nodes in order to reduce the depth of LM. Additionally the effect was not so dramatic as I hoped. In one of the bad cases I discovered a LM depth of 348. After the optimization it came down to 131. But the additional memory requirement of over 200MBytes breaks every rout-

ers memory limit.

5.4 Further Work

The problem of the long list depth leads to high lookup counts in a few ACL-Mtrie+. The problem arises when more specific statements append their lines to a less specific line. Then all lines get appended to the same list. An idea might be to avoid the situation that a less specific statement is preceding more specific ones. Of course the ACL semantic mustn't be changed.

The algorithm which analyzes the ACL and selects the appropriate node sequences is quite simple. Perhaps with a more complex analysis the lookup count and memory usage could be lowered further.

6.0 Concluding Remarks

This project analysed the ACL-Mtrie+ approach. The goal of the ACL-Mtrie+ simulation was to keep the lookup count and the memory usage low. The depth of the originally long linked list of match nodes is reduced by inserting additional demux nodes before it. Specifically the fact that we demux on the protocol first gives us the opportunity to take advantage of the fact that the traffic pattern consists mostly of established TCP packets. This resulted in a low lookup count.

The ACL-Mtrie+ is compared to the IOS traffic filter an improvement in terms of lookup counts. But it will never reach the efficiency of the CAM approach. That is because all other protocols than the optimized established TCP have a lookup count of more than one but the CAM approach is constantly 1.

However the MPE combines the traffic filtering and routing in one device. That's a great advantage because now the routing subtree has access not only to the IP destination address but all the other fields within the packet header too. Furthermore the forwarding rate with the ACL-Mtrie+ is adequate for most situations. It's performance will increase with time because of additional optimizations and new forms.

7.0 Acknowledgments

I'd like to thank Andy Bechtolsheim, David Cheriton, Fusun Ertemalp, Hugh Holbrook, Isabelle Bertin-Bailly, Lorenz Redlefson, Matt Zelesko, Steve Deering and many other people in the GSG for their support and assistance.

I very appreciated the opportunity to perform my diploma thesis at such an exciting company like Cisco Systems Inc., San Jose, CA. Special thanks here to Andy Bechtolsheim and David Cheriton who made that possible.

Appendix A: Mtrie

The Mtrie is a quite simple way how to do all kind of lookups within routers. Mostly the Mtrie is used for IP destination lookups. Used for the IP destination lookup, the depth of the classical

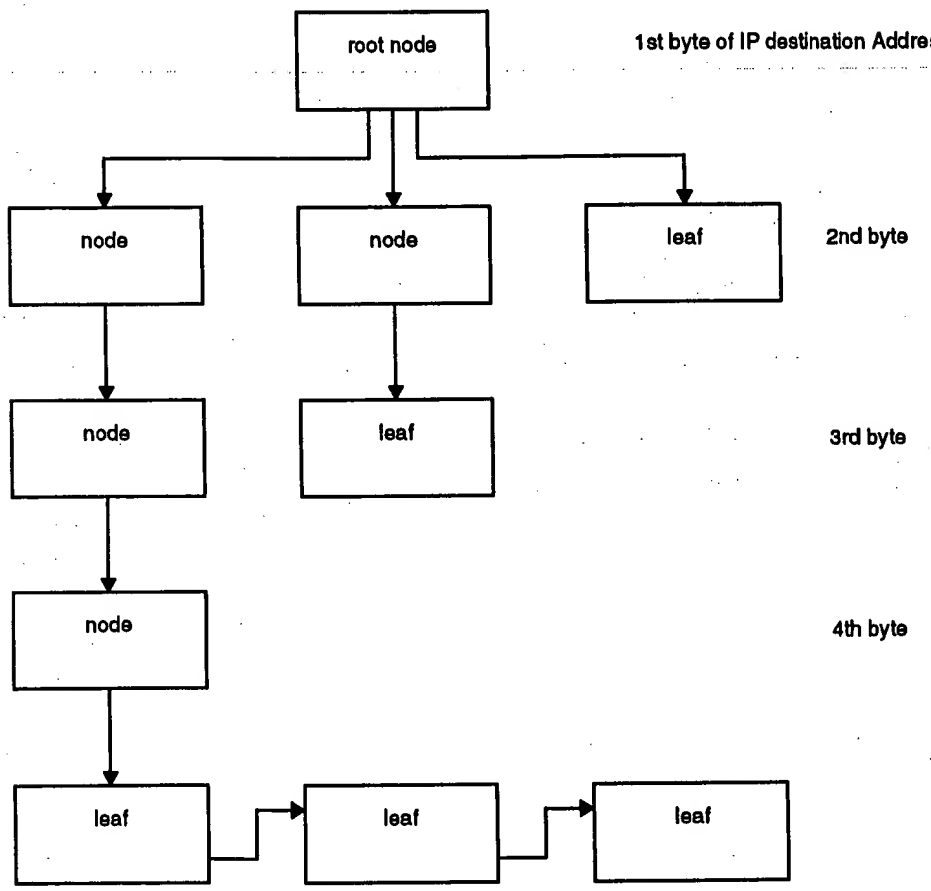

Figure-32 Classical Mtrie

Mtrie is bounded to 5 (e.g. 4 nodes & one leaf). The Mtrie serves for looking up the next hop (e.g. the output interface) based on the destination address of the given IP-packet. A destination address lookup operates after the "longest match" scheme and is performed in the following way:

1. The value of the first byte of the destination address of a received packet is used as index into the pointer-array stored in the root node.
2. If the pointer read out has the LSB set[1], the element the pointer (with unsetting the LSB) is referencing is a leaf, otherwise the element referenced is another node.
   - If the element is another node, the same operation as in 1. is performed with the second address byte.

1.The LSB can be used for distinguishing between Mtrie nodes and Mtrie leafs. For obtaining a valid memory address, the LSB of the concerned pointer has to cleared.

- If the element is a leaf, the output interface ID (and the next hop) is read out of the leaf and the packet is forwarded to the output interface.

Let's consider a simple example for a routing table lookup. Assuming that the class A network 34.0.0.0 is reachable through the interface with the ID 3. So in the first step the root's array element 34 is read. The pointer read with this operation is referring to a leaf in which the output interface ID 3 can be found.

This existing Mtrie offers furthermore the facility to "remember" previous routes. So if an existing route is overwritten by a more specific one e.g. a new level of nodes is added to a certain branch of the tree, the former routing decision is stored in a chained list and can be recovered if the more specific route is removed again.

Appendix B: Class description of the ACL-Mtrie+ Simulation

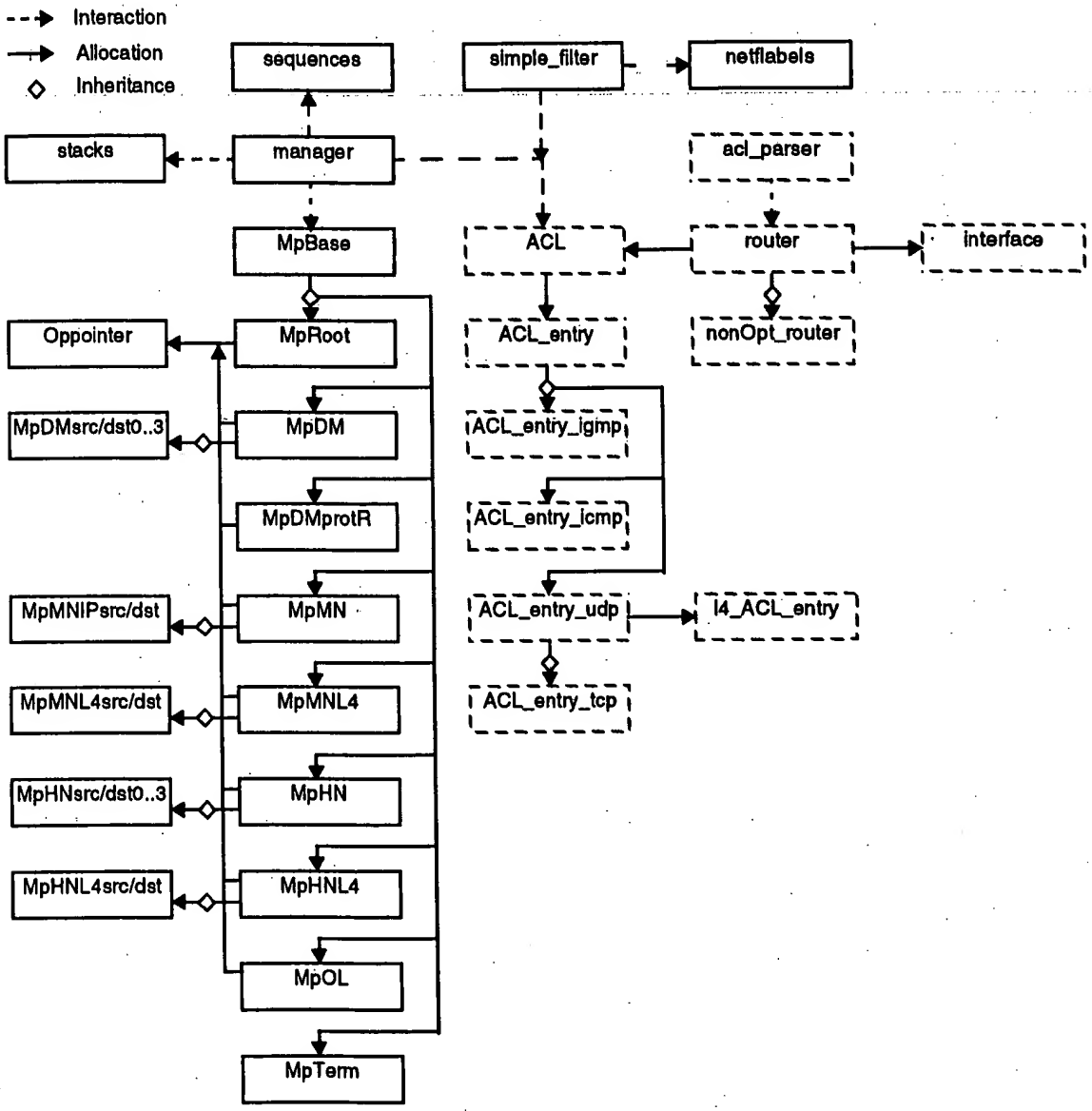

Figure-33 Object Model ACL-Mtrie+ Simulator

The Figure-33 shows the object Model of the ACL-Mtrie+ Simulation. The boxes with the broken frame line represent objects which we reused for this projects. Their definition and code is explained in [3].

We defined some mtrie node aliases in Table-6. This aliases basically label some opcodes with more intuitive shortcuts that will serve as C++ class-names.

Opcode	Shortcut (alias)	Description
8	MpMNprot	Match node on the protocol field[a]
9	MpMNIPsrc	Match node on the IP source address
10	MpMNIPdst	Match node on the IP destination address
14	MpMNest	Match node on the established flag (syn-bit) of a TCP-PDU[a]
35	MpDMprot	Demux on protocol type
36	MpDMIPsrc0	Demux on first byte of IP source address
37	MpDMIPsrc1	Demux on second byte of IP source address
38	MpDMIPsrc2	Demux on third byte of IP source address
39	MpDMIPsrc3	Demux on fourth byte of IP source address
40	MpDMIPdst0	Demux on first byte of IP destination address
41	MpDMIPdst1	Demux on second byte of IP destination address
42	MpDMIPdst2	Demux on third byte of IP destination address
43	MpDMIPdst3	Demux on fourth byte of IP destination address
64	MpMNL4dst	Match node on the layer 4 destination port
65	MpMNL4src	Match node on the layer 4 source port
66	MpDMprotR	Reduced demux on IP, UDP, TCP and established TCP
67	MpDMIPdstR	Demux on the 24bit prefix of the destination address
68	MpHNIPsrc0	Hash on first byte of IP source address
69	MpHNIPsrc1	Hash on second byte of IP source address
70	MpHNIPsrc2	Hash on third byte of IP source address
71	MpHNIPsrc3	Hash on fourth byte of IP source address
72	MpHNIPdst0	Hash on first byte of IP destination address
73	MpHNIPdst1	Hash on second byte of IP destination address
74	MpHNIPdst2	Hash on third byte of IP destination address
75	MpHNIPdst3	Hash on fourth byte of IP destination address
76	MpHNL4src	Hash on the layer 4 source port
77	MpHNL4dst	Hash on the layer 4 destination port
78	MpMNbIP	Match on both IP addresses
79	MpMNbIPL4	Match on both IP addresses and on both layer 4 ports
80	MpDML4src	Demux on the layer 4 source port
81	MpDML4dst	Demux on the layer 4 destination port
127	mnaOL	Output interface leaf

Table-6 Overview of shortcut aliases

Next we explain shortly the used classes and functions of the simulation. For each class the template of Table-7 is used for giving an overview of the class.

Class name
source- and header files
derived from...
public interface
protected interface
names of classes that are used by this class

Table-7 Template for Class Description

B.1 Oppointer Class

Oppointers	
mtrie+_nodes.{hlcc}	
-	

Oppointers();
int opcode() const;
void opcode(int opcode);
MpBase* address() const;
void address(MpBase* paddr);
	
	read/write the opcode
	
	read/write the address

-	
-	

Table-8 Oppointers Class

This class is the core of the node classes and represents the idea of the oppointer. The address points to the next node and the opcode describes its type.

B.2 Node Classes

The MpBase class defines several virtual functions which are then replaced by the derived classes. This simplifies the access to the oppointers and other data in the lookup function.

MpBase	
mtrie+_nodes.{hlcc}	
-	

MpBase(int nodeType);
virtual int opcode(int idx)
virtual void opcode(int idx, int opcode)
virtual MpBase* address(int idx)
virtual void address(int idx, MpBase* const paddr)
virtual ipaddr pattern()
virtual ipaddr addrMask()
virtual uint2 start()
virtual uint2 end()
int nodeType() const;
long refCount() const;
void refCount(bool up);
void refCount(long n);
long childCount() const;
void childCount(bool up);
void childCount(long n);
	sets the node type at construction
	read/write the opcode and address of the oppointers
	
	read/write the pattern and mask of the match nodes
	
	read the start and end port number of the layer 4 match nodes
	get the node type of this node
	read/write the reference- and child-counts of the nodes

-	
Oppointers	

Table-9 MpBase Class

Because all nodes have an array of oppointer, the public interfaces of all derived classes is a subset of the one in the MpBase class with additional individual methods. Therefore they look very similar and are not listed here. All node classes are declared in the mtrie+_nodes.{hlcc}.

B.3 Manager Class

Managers	
managers.{hlcc}	
-	
Managers(acl *pacl);	instantiate an manager object with an ACL
// preprocessing methods	
void analyze_acl();	analyze the ACL and select the node sequences
void init_demux_l4(MpBase *node, acl_entry *pselect);	initialize the demux layer 4 node
// building methods	
void add_acl();	this starts the ACL-Mtrie+ building
MpBase* new_node(acl_entry* pentry, Sequences* pseq, int seqIdx);	acquire a new node
MpBase* new_node(acl_entry* pentry, FieldTypes ft);	
MpBase* create_line(acl_entry *pentry, Sequences *pseq);	create a line of match nodes for the linked list of match nodes
bool match_line(MpBase *porg, MpBase *pcur);	check if both match lines match
int append_line(MpBase *porg, MpBase *pcur);	append a match line to a linked list of match nodes
int linkEnd(MpBase* p);	get the size of the oppointer array, the last index
uint1 get_byte(acl_entry *pentry, Sequences* pseq, int seqIdx);	get the current byte of the mask according to the cur rent node
MpBase* replicate_list(MpBase *p);	replicate a whole linked list of match nodes
MpBase* replicate_subtree(MpBase *p);	replicate a whole subtree in the ACL-Mtrie+
// revise methods	
bool redundant(MpBase* p);	check if this node is redundant
void purge();	purge the ACL-Mtrie+ from all redundant nodes
// debug methods	
void print_list(MpBase *pb);	print the linked list of match nodes
// get methods	
MpBase* root() const;	get the root node address
-	
-	

Table-10 Managers Class

This class is responsible for building the ACL-Mtrie+ data structure.

The add_acl() calls the analyze_acl() method to select the appropriate node sequences, then processes all entries in an ACL and appends all necessary nodes and lists.

B.4 Netflabels Class

Netflabels	
netflabels.{hlcc}	
-	
Netflabels(); void clear(); ipaddr& operator[](int idx);	 clear the packet label read/write the packet label array
-	
-	

Table-11 Netflabels Class

This class is the implementation of the packet label and serves as a container for the test packet labels in the function test_algo().

B.5 Simple_Filter Class

Simple_filters	
simple_filters.{hlcc}	
-	
Simple_filters(acl *pa); bool filter(Netflabels &plabel, int &entry_idx);	 get the filtering decision and which entry was hit

bool matchIp(Netflabels &pl);
bool matchProt(Netflabels &pl);
bool matchEst(Netflabels &pl);
bool matchPort(Netflabels &pl);
	try to match on different qualifiers of an entry

-	

Table-12 Simple_filters Class

This class is used to do the verification of the ACL-Mtrie+ filtering decisions. The current test packet label is sequentially compared to the statements in the ACL object. As soon as a match occurs, the process is stopped and the decision returned. If none of the statements match then the default is drop by convention and the entry index is set to zero.

B.6 Sequences Class

This class is needed for describing and handling the node sequences. The 'Appendix D: Sequence

List Configurations' shows all current node sequences.

Sequences	
sequences.{hlcc}	
-	

enum FieldTypes {
end, stop, leaf, demux_protocol, demux_protR,
demux_l4_dst, match_ip_src, match_ip_dst,
match_ip, match_ipl4, demux_ip_src0,
demux_ip_src1, demux_ip_src2, demux_ip_src3,
demux_ip_dst0, demux_ip_dst1, demux_ip_dst2,
demux_ip_dst3, hash_ip_src0, hash_ip_src1,
hash_ip_src2, hash_ip_src3, hash_ip_dst0,
hash_ip_dst1, hash_ip_dst2, hash_ip_dst3,
demux_l4_src0, demux_l4_src1, demux_l4_dst0,
demux_l4_dst1, match_l4_src, match_l4_dst,
hash_l4_dst, match_l4_est, match_protocol,
match_list
};
Sequences();
Sequences(const char* filename);
~Sequences();
 	
bool init(const char* filename);
const int& position();
const FieldTypes& get_first();
const FieldTypes& get_next();
const FieldTypes& get_cur();
const FieldTypes& get_pos(const int& index);
	all the different node types
	
	read a textfile and initialize a node sequence array
	set/get the current position in the array

-	
-	

Table-13 Sequences Class

B.7Stacks Class

Stacks	
stacks.{hlcc}	
-	

Stacks();
void push(StackElements *pse);
StackElements* pop();
void print();
	
	push a StackElements object on the stack
	pop it from the stack

-	
-	

Table-14 Oppointers Class

This class is used by the replicate_subtree() method in the managers class.

StackElements	
stacks.{hlcc}	
-	

StackElements(MpBase *p, int n);
void node(MpBase *p);
MpBase* node();
void linkIdx(int n);
int linkIdx();
	
	read/write the node address
	
	read/write the link index

-	
-	

Table-15 Oppointers Class

B.8 Functions

The following functions are found in the main module mtrie+test.cc:

void test_algo(Managers *m, acl *pa)
bool lookup(MpBase *p, Netflabels& tst)
void reset_statistic()
void print_statistics()
void print_statistics_excel()

The test_algo() executes the verification of the ACL-Mtrie+. First for every statement in the ACL object several test packet labels are created. Then they are processed through the ACL-Mtrie+ and the Simple_filters object. If the results are not the same, an error message is created and the affected ACL statement and test packet label is printed out. This helps to track down a wrong tree structure.

The lookup() is used in the test_algo() for getting the filtering decision in the ACL-Mtrie+. In this function the MPE lookup is emulated. As we know each node holds an array of oppointers. The oppointers are divided into the opcode and the address. For selecting the correct next oppointer in the array, the algorithm performs an operation determined by the opcode on the packet label. Then the algorithm advances to the next node until to a leaf node. Finally the filtering decision according to the leaf node is returned.

The print_statistics() function produce the following information of the ACL-Mtrie+ data structure: router configuration filename, current ACL number, number of errors, quantity of each used node type, quantity of memory space of each used node type, total used memory space, maximum depth of the lists for each protocol, minimal and maximal lookup count, mean lookup count for each protocol. The print_statistics_excel() produces the statistics in an excel readable form.

Appendix C: Programs for the analysis of ACLs and traces

The following programs were used for the analysis:

aclStat filename
extract filename extension begin# end#

The program aclStat produces several statistics about the ACL in a router configuration file. We used it in our analysis for the statistics of the layer 4 port numbers and of the protocols. The input file is a list of router configuration files.

The program extract was used to count the number of packets sorted by the protocol type. The trace files are usually fragmented. Therefore their filename is labeled additionally with a number. The *begin#* and *end#* instruct the extract program which trace files to process.

Appendix D: Node Sequence Configurations

For each branch in the reduced protocol node we use different node sequence lists (Table-14). Within each branch we adapt the node sequence to the results of the ACL analysis (Table-15).

seq_l4all.txt	seq_all.txt	seq_protall.txt
demux_protR demux_l4_dst demux_ip_src0 demux_ip_src1 demux_ip_src2 demux_ip_src3 demux_ip_dst0 demux_ip_dst1 demux_ip_dst2 demux_ip_dst3 match_list stop	demux_protR demux_ip_src0 demux_ip_src1 demux_ip_src2 demux_ip_src3 demux_ip_dst0 demux_ip_dst1 demux_ip_dst2 demux_ip_dst3 match_list stop	demux_protR demux_protocol demux_ip_src0 demux_ip_src1 demux_ip_src2 demux_ip_src3 demux_ip_dst0 demux_ip_dst1 demux_ip_dst2 demux_ip_dst3 match_list stop
seq_l4few.txt	seq_few.txt	seq_protfew.txt
demux_protR demux_l4_dst demux_ip_src0 match_list stop	demux_protR match_list stop	demux_protR demux_protocol match_list stop
	seq_one.txt	
	demux_protR leaf stop	

Table-16 Node Sequences

conditions	UDP/TCP	established TCP	IP
allow all traffic		seq_one.txt	
less than 7 state-ments	seq_l4few.txt	seq_few.txt	seq_protfew.txt
other	seq_l4all.txt	seq_all.txt	seq_protall.txt
linked list of match node sequence	seq_ipMatch.txt	seq_ipl4Match.txt	seq_ipMatch.txt

Table-17 Which Node Sequence?

Appendix E: Example in how Subdivide the Lists

Here we illustrate the process of how the depth of a list is decreased by inserting additional nodes and therefore subdividing the list into smaller ones. Here is the example ACL:

```
A  access-list 142  permit udp  152.163.82.28 0.0.0.0      any                 eq 161
B  access-list 142  permit udp  152.163.198.0 0.0.0.255    198.81.9.35  0.0.0.0 eq 7
C  access-list 142  deny   udp  any                        any                 eq 2049
D  access-list 142  permit udp  any                        152.163.200.2 0.0.0.0 gt 1023
E  access-list 142  permit udp  any                        152.163.200.3 0.0.0.0 gt 1023
F  access-list 142  permit udp  152.163.7.25  0.0.0.0      any                 gt 1023
G  access-list 142  permit udp  any                        152.163.82.28 0.0.0.0 gt 1023
H  access-list 142  deny   udp  any                        any                 gt 33553
J  access-list 142  permit udp  152.163.0.0   0.0.255.255  any                 gt 33420
K  access-list 142  permit udp  198.81.0.0    0.0.31.255   any                 gt 33420
```

In the following discussion the lines of the statements are referenced by their character.

sector	demux on 1. src byte	demux on 3. dst byte	demux on 4. dst byte
1024..33420: D E F G	152: D E F G	200: D E F	2: ***D*** ***F***
			3: ***E*** ***F***
			other: ***F***
		82: E F	28: ***E*** ***F***
			other: ***E***
		other: ***F***	
	other: D E G	200: D E	2: ***D***
			3: ***E***
		82: ***G***	

Table-18 Subdivide the List

First we demux on the layer 4 destination port. This results in 6 different sectors:

```
    7: B
  161: A
 2049: C (D..G are not appended because they match all on IP address of 3)
1024..33420: D..G (port number 2049 is excluded from this range)
33421..33553: D..G, J..K
33554..65535: C..K
```

The sectors 7, 161 and 2049 have now only one line. In the other sectors we can decrease the list depth further by inserting IP address demux nodes. Let's have a closer look at the sector 1024..33420. The Table-14 illustrates how the subdividing is done. The leaf position is indicated by the bold and italic style of the statement characters. We see that the list depth of ten is reduced to one or two lines.

Appendix F: Analysis of Router Configurations

This is the complete analysis of the router configuration database. In the first column we have the filename of the router configuration file, then the ACL number. The sum column shows the memory requierement for this ACL-Mtrie+. The list columns show the depth of the linked list of match nodes. All the following columns are for counting the lookup steps in different aspects.

filename	acl_nr	sum [bytes]||	list TCP	list UDP	list ETCP	list IP	min	max	mean TCP	mean UDP	mean ETCP	mean IP	expected case	adversary case
	100	"821,456"	1	1	1	1	3	11	6.8	8.7	8.7	3.2	8.8	6.8
	102	"574,056"	2	2	0	1	3	11	5.0	4.3	0.0	8.8	5.1	8.0
	103	"710,432"	5	1	8	1	4	14	8.2	9.4	7.3	9.9	7.3	8.7
	104	"792,200"	2	1	2	1	2	10	8.8	8.9	3.0	3.7	3.1	6.1
	110	"1,184"	0	0	0	3	4	6	0.0	0.0	0.0	5.3	5.3	5.3
	111	"1,084"	0	0	0	1	3	3	0.0	0.0	0.0	3.0	3.0	3.0
	112	"1,104"	0	0	0	1	4	4	0.0	0.0	0.0	4.0	4.0	4.0
	90	"15,672"	0	0	0	2	4	7	0.0	0.0	0.0	6.3	6.3	6.3
	150	"30,016"	0	0	0	1	4	8	0.0	0.0	0.0	6.8	6.8	6.8
	90	"10,000"	0	0	0	1	4	7	0.0	0.0	0.0	6.1	6.1	6.1
	92	"9,936"	0	0	0	1	4	7	0.0	0.0	0.0	6.1	6.1	6.1
	100	"14,448"	0	0	0	1	5	8	0.0	0.0	0.0	7.2	7.2	7.2
	127	"265,560"	0	2	0	1	3	6	4.7	0.0	0.0	3.0	4.4	3.8
	150	"30,016"	0	0	0	1	4	8	0.0	0.0	0.0	6.8	6.8	6.8
	111	"1,159,552"	2	1	0	1	3	11	7.1	9.7	0.0	3.0	7.7	6.6
	121	"588,176"	3	1	0	1	3	8	7.3	4.8	0.0	3.0	5.9	5.0
	101	"908,704"	18	1	0	4	1	19	4.0	10.6	1.0	4.7	1.1	5.1
	102	"657,320"	19	1	0	4	1	20	4.0	11.1	1.0	4.7	1.1	5.2
	101	"818,880"	5	1	0	5	1	10	4.0	8.1	1.0	5.6	1.1	4.7
	102	"820,872"	12	1	0	5	1	19	4.0	10.4	1.0	7.0	1.1	5.6
	101	"765,144"	23	1	0	4	1	22	4.0	10.3	1.0	4.7	1.1	5.0
	102	"787,184"	12	1	0	4	1	19	4.0	10.5	1.0	4.7	1.1	5.0
	101	"795,384"	5	1	0	1	1	10	4.0	8.1	1.0	6.4	1.1	4.9
	102	"803,328"	12	1	0	1	1	19	4.0	10.5	1.0	6.4	1.1	5.5
	101	"789,216"	5	1	0	4	1	10	4.0	8.1	1.0	4.7	1.1	4.4

Table-19 Analysis of Router Configurations

filename	acl_nr	sum [bytes]fi	list TCP	list UDP	list ETCP	list IP	min	max	mean TCP	mean UDP	mean ETCP	mean IP	expected case	adversary case
	102	"797,160"	12	1	0	4	1	19	4.0	10.5	1.0	4.7	1.1	5.1
	101	"850,704"	11	1	0	3	1	15	4.0	8.2	1.0	4.2	1.1	4.3
	102	"2,097,784"	12	1	0	3	1	19	4.0	10.2	1.0	4.2	1.1	4.9
	101	"773,712"	5	1	0	4	1	10	4.0	8.2	1.0	4.7	1.1	4.5
	102	"549,152"	34	1	0	4	1	39	4.0	18.3	1.0	4.7	1.2	7.0
	101	"827,960"	7	1	0	6	1	11	4.0	8.2	1.0	6.5	1.1	4.9
	102	"1,151,776"	12	1	0	5	1	19	4.0	10.6	1.0	7.2	1.1	5.7
	101	"814,304"	5	1	0	4	1	10	4.0	8.3	1.0	7.3	1.1	5.1
	102	"601,312"	19	1	0	4	1	26	4.0	10.6	1.0	6.9	1.1	5.8
	101	"757,216"	3	1	0	15	1	23	4.0	7.0	1.0	9.9	1.1	5.5
	102	"926,968"	7	1	0	2	1	14	4.0	6.4	1.0	7.6	1.1	4.7
	163	"123,200"	0	0	0	8	5	16	0.0	0.0	0.0	8.9	8.9	8.9
	11	"1,184"	0	0	0	3	4	6	0.0	0.0	0.0	5.1	5.1	5.1
	102	"99,952"	0	0	0	1	3	11	0.0	0.0	0.0	10.8	10.8	10.8
	100	"811,640"	2	1	1	1	3	11	7.2	9.3	9.2	3.0	9.2	7.2
	101	"1,078,520"	1	1	2	1	3	11	8.2	10.0	7.9	3.0	7.9	7.3
	100	"899,560"	1	2	0	1	3	11	10.2	9.4	0.0	3.0	9.2	7.5
	100	"811,640"	2	1	1	1	3	11	7.2	9.3	9.2	3.0	9.2	7.2
	101	"1,078,520"	1	1	2	1	3	11	8.2	10.0	7.9	3.0	7.9	7.3
	100	"899,560"	1	2	0	1	3	11	10.2	9.4	0.0	3.0	9.2	7.5
	101	"654,824"	1	1	1	1	3	11	4.5	11.0	9.9	3.0	9.9	7.1
	102	"872,624"	1	1	2	1	2	11	4.5	11.0	3.0	3.0	3.1	5.4
	101	"26,568"	0	0	1	1	3	10	0.0	0.0	10.0	3.0	10.0	6.5
	102	"485,856"	1	0	0	1	3	11	0.0	11.0	0.0	3.0	9.4	7.0
	101	"429,448"	2	0	1	1	3	11	0.0	9.6	9.6	3.0	9.6	7.4
	102	"403,048"	1	0	1	1	3	11	0.0	10.3	8.4	3.0	8.4	7.2
	2	"1,104"	0	0	0	1	3	5	0.0	0.0	0.0	4.0	4.0	4.0
	100	"621,296"	1	1	3	4	2	11	11.0	11.0	3.5	6.7	3.6	8.0
	150	"535,936"	1	1	1	1	2	5	4.2	4.2	2.5	4.0	2.5	3.7
	100	"625,208"	1	1	3	1	2	11	11.0	11.0	3.5	4.0	3.6	7.4
	102	"657,984"	2	3	2	1	2	11	4.7	8.2	3.3	4.0	3.4	5.1
	150	"535,960"	1	1	1	1	2	5	4.3	4.2	2.5	4.0	2.5	3.8
	101	"4,730,016"	2	1	1	1	2	11	9.4	10.5	2.5	8.4	2.6	7.7
	100	"1,114,392"	387	3	1	1	2	374	9.2	102.2	2.5	7.6	3.4	30.4
	101	"830,360"	14	1	2	1	2	19	10.1	8.2	3.0	3.5	3.1	6.2
	100	"934,272"	7	0	5	29	2	38	0.0	10.1	4.5	15.3	4.6	10.0
	101	"1,065,240"	1	1	1	19	7	27	10.7	10.4	9.1	15.8	9.1	11.5
	102	"331,496"	4	0	0	1	3	13	0.0	10.9	0.0	3.5	9.4	7.2
	100	"1,624,664"	674	18	1	3	2	679	8.5	288.6	2.5	9.1	4.9	77.2
	101	"1,129,024"	19	2	1	1	2	27	10.2	10.1	2.5	8.9	2.7	7.9
	100	"883,152"	68	1	0	5	7	16	10.1	10.3	0.0	11.1	10.3	10.5
	101	"2,448,968"	71	1	1	1	2	79	11.0	18.2	2.0	10.5	2.2	10.4
	100	"791,160"	16	1	0	5	7	16	10.3	10.0	0.0	11.3	10.3	10.6
	101	"3,456,176"	70	1	1	1	2	52	11.0	12.5	2.0	10.5	2.2	9.0
	100	"292,184"	1	0	1	1	2	8	0.0	4.0	2.5	7.3	2.5	4.8
	120	"722,104"	5	2	1	1	2	11	5.0	7.8	2.5	8.9	2.6	6.1
	121	"1,160"	0	0	0	3	3	6	0.0	0.0	0.0	4.2	4.2	4.2
	101	"529,576"	1	1	1	1	2	4	4.0	4.0	2.0	3.5	2.0	3.4
	120	"1,386,808"	4	8	1	1	2	13	10.2	8.4	2.0	10.3	2.1	7.7
	121	"1,136"	0	0	0	2	3	5	0.0	0.0	0.0	3.8	3.8	3.8

Table-19 Analysis of Router Configurations

filename	acl_nr	sum [bytes]fi	list TCP	list UDP	list ETCP	list IP	min	max	mean TCP	mean UDP	mean ETCP	mean IP	expected case	adversary case
	101	"529,576"	1	1	1	1	2	4	4.0	4.0	2.0	3.5	2.0	3.4
	120	"1,215,416"	3	6	1	1	2	13	10.2	8.5	2.0	10.3	2.1	7.7
	121	"1,136"	0	0	0	2	3	5	0.0	0.0	0.0	3.8	3.8	3.8
	103	"682,976"	1	1	4	2	2	11	9.3	9.3	4.0	3.8	4.1	6.6
	104	"682,400"	1	1	4	2	2	11	9.3	9.4	4.0	3.8	4.1	6.6
	101	"637,800"	1	1	1	1	2	9	5.3	6.1	2.0	3.0	2.1	4.1
	110	"626,624"	1	1	0	1	1	11	5.4	5.7	1.0	3.0	1.1	3.8
	120	"299,224"	1	0	0	1	5	10	0.0	6.5	0.0	5.1	6.3	5.8
	103	"695,360"	1	1	4	2	2	11	9.3	9.2	4.0	3.8	4.1	6.6
	104	"698,880"	1	1	4	2	2	11	9.3	9.3	4.0	3.8	4.1	6.6
	48	"1,104"	0	0	0	1	4	4	0.0	0.0	0.0	4.0	4.0	4.0
	110	"718,584"	5	2	0	5	3	11	8.7	7.4	0.0	5.5	7.9	7.2
	120	"877,064"	2	1	0	1	3	9	7.9	8.1	0.0	3.0	7.5	6.4
	130	"596,520"	1	1	0	1	3	10	8.9	8.3	0.0	3.0	8.1	6.7
	48	"1,104"	0	0	0	1	4	4	0.0	0.0	0.0	4.0	4.0	4.0
	110	"718,584"	5	2	0	5	3	11	8.7	7.4	0.0	5.5	7.9	7.2
	120	"877,064"	2	1	0	1	3	9	7.9	8.1	0.0	3.0	7.5	6.4
	130	"596,520"	1	1	0	1	3	10	8.9	8.3	0.0	3.0	8.1	6.7
	150	"922,064"	2	2	1	12	2	12	8.5	7.8	2.5	7.1	2.6	6.5
	50	"35,912"	0	0	0	1	5	8	0.0	0.0	0.0	6.0	6.0	6.0
	50	"35,912"	0	0	0	1	5	8	0.0	0.0	0.0	6.0	6.0	6.0
	100	"124,768"	0	0	0	348	6	259	0.0	0.0	0.0	94.4	94.4	94.4
	101	"977,144"	0	0	0	2	7	12	0.0	0.0	0.0	11.1	11.1	11.1
	3	"18,240"	0	0	0	1	6	7	0.0	0.0	0.0	7.0	7.0	7.0
	1	"7,080"	0	0	0	1	7	7	0.0	0.0	0.0	7.0	7.0	7.0
	101	"33,672"	0	0	0	1	8	10	0.0	0.0	0.0	9.4	9.4	9.4
	101	"559,016"	0	0	0	1	6	11	0.0	0.0	0.0	10.6	10.6	10.6
	101	"562,304"	0	0	0	1	6	11	0.0	0.0	0.0	10.6	10.6	10.6
	111	"680,832"	1	1	1	3	2	11	11.0	11.0	2.5	4.5	2.7	7.3
	121	"707,568"	1	1	1	3	2	11	10.2	11.0	2.5	4.3	2.6	7.0
	130	"584,016"	1	1	1	1	2	11	4.0	11.0	2.0	3.0	2.1	5.0
	100	"23,188"	0	0	0	1	3	9	0.0	0.0	0.0	8.6	8.6	8.6
	101	"99,892"	0	0	0	1	3	9	0.0	0.0	0.0	8.6	8.6	8.6
	102	"23,144"	0	0	0	1	3	9	0.0	0.0	0.0	8.6	8.6	8.6
	103	"99,992"	0	0	0	1	3	9	0.0	0.0	0.0	8.6	8.6	8.6
	100	"160,872"	0	0	0	1	9	11	0.0	0.0	0.0	10.7	10.7	10.7
	100	"603,824"	1	1	0	1	3	10	4.0	4.5	0.0	9.2	4.7	5.9
	104	"572,800"	2	1	0	1	3	8	5.4	6.9	0.0	3.0	5.7	5.1
	100	"362,688"	0	1	0	1	4	10	4.0	0.0	0.0	9.4	4.9	6.7
	101	"37,168"	0	0	0	1	9	9	0.0	0.0	0.0	9.0	9.0	9.0
	108	"114,544"	0	0	0	1	9	9	0.0	0.0	0.0	9.0	9.0	9.0
	111	"108,328"	0	0	0	1	9	9	0.0	0.0	0.0	9.0	9.0	9.0
	113	"371,536"	1	0	0	1	5	9	0.0	5.0	0.0	9.0	5.8	7.0
	100	"400,008"	0	1	0	1	4	10	4.0	0.0	0.0	9.4	4.9	6.7
	101	"129,168"	0	0	0	1	9	9	0.0	0.0	0.0	9.0	9.0	9.0
	102	"58,792"	0	0	0	1	9	9	0.0	0.0	0.0	9.0	9.0	9.0
	108	"265,400"	0	1	0	4	3	7	4.0	0.0	0.0	5.0	4.2	4.5
	101	"67,096"	0	0	0	2	4	11	0.0	0.0	0.0	10.6	10.6	10.6
	101	"67,096"	0	0	0	2	4	11	0.0	0.0	0.0	10.6	10.6	10.6
	108	"1,090,600"	1	0	0	1	7	11	0.0	8.1	0.0	10.0	8.5	9.1

Table-19 Analysis of Router Configurations

filename	acl_nr	sum [bytes]ñ	list TCP	list UDP	list ETCP	list IP	min	max	mean TCP	mean UDP	mean ETCP	mean IP	expected case	adversary case
	21	"12,192"	0	0	0	1	5	7	0.0	0.0	0.0	6.4	6.4	6.4
	25	"17,840"	0	0	0	1	5	7	0.0	0.0	0.0	6.7	6.7	6.7
	31	"1,136"	0	0	0	3	3	6	0.0	0.0	0.0	4.5	4.5	4.5
	41	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	105	"1,445,584"	3	1	1	5	5	12	7.4	10.7	9.9	7.7	8.9	8.9
	131	"469;184"	1	0	0	1	4	11	0.0	4.5	0.0	10.4	5.7	7.5
	15	"1,104"	0	0	0	1	4	4	0.0	0.0	0.0	4.0	4.0	4.0
	22	"15,032"	0	0	0	1	4	6	0.0	0.0	0.0	6.0	6.0	6.0
	31	"1,104"	0	0	0	1	3	5	0.0	0.0	0.0	4.0	4.0	4.0
	31	"1,104"	0	0	0	1	3	5	0.0	0.0	0.0	4.0	4.0	4.0
	33	"1,176"	0	0	0	4	3	8	0.0	0.0	0.0	5.2	5.2	5.2
	34	"1,152"	0	0	0	3	3	7	0.0	0.0	0.0	4.8	4.8	4.8
	51	"1,104"	0	0	0	1	3	5	0.0	0.0	0.0	4.0	4.0	4.0
	52	"1,104"	0	0	0	1	3	5	0.0	0.0	0.0	4.0	4.0	4.0
	53	"1,104"	0	0	0	1	3	5	0.0	0.0	0.0	4.0	4.0	4.0
	22	"37,952"	0	0	0	1	5	7	0.0	0.0	0.0	6.1	6.1	6.1
	102	"521,760"	1	0	0	71	5	42	0.0	10.8	0.0	19.2	12.5	15.0
	22	"24,992"	0	0	0	1	5	7	0.0	0.0	0.0	6.0	6.0	6.0
	33	"29,792"	0	0	0	1	5	7	0.0	0.0	0.0	6.8	6.8	6.8
	102	"1,362,088"	1	1	2	69	2	45	7.9	11.6	3.0	22.0	3.2	11.1
	131	"284,328"	1	0	0	2	3	5	0.0	4.5	0.0	4.0	4.4	4.3
	21	"1,064"	0	0	0	1	3	3	0.0	0.0	0.0	3.0	3.0	3.0
	31	"29,272"	0	0	0	1	5	7	0.0	0.0	0.0	6.4	6.4	6.4
	104	"680,760"	1	4	0	7	5	12	9.7	11.3	0.0	6.8	10.0	9.2
	104	"676,808"	1	4	0	5	4	12	9.6	11.3	0.0	6.6	10.0	9.1
	131	"22,144"	0	0	0	29	5	36	0.0	0.0	0.0	12.1	12.1	12.1
	31	"25,008"	0	0	0	1	5	7	0.0	0.0	0.0	6.9	6.9	6.9
	104	"676,672"	2	2	1	3	4	12	4.5	11.4	10.0	7.9	10.0	8.4
	104	"840,464"	7	1	1	4	5	17	10.1	11.5	9.6	7.2	9.7	9.8
	131	"515,656"	1	0	1	1	2	11	0.0	11.0	2.0	9.7	2.1	7.8
	10	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.1	4.1	4.1
	23	"1,104"	0	0	0	1	3	5	0.0	0.0	0.0	4.0	4.0	4.0
	31	"1,200"	0	0	0	5	3	9	0.0	0.0	0.0	5.7	5.7	5.7
	105	"815,664"	2	1	0	3	4	12	4.0	11.3	0.0	7.8	7.3	7.7
	21	"14,320"	0	0	0	1	5	7	0.0	0.0	0.0	6.4	6.4	6.4
	31	"12,064"	0	0	0	1	5	7	0.0	0.0	0.0	6.4	6.4	6.4
	41	"1,152"	0	0	0	3	3	7	0.0	0.0	0.0	4.8	4.8	4.8
	104	"885,504"	1	1	0	15	4	22	4.0	11.0	0.0	12.2	7.6	9.1
	105	"266,480"	1	0	0	3	3	6	0.0	4.0	0.0	4.5	4.1	4.2
	107	"348,032"	1	0	2	3	2	11	0.0	11.0	3.0	8.3	3.1	7.4
	115	"1,152"	0	0	0	3	3	7	0.0	0.0	0.0	4.8	4.8	4.8
	141	"437,936"	1	0	0	5	5	11	0.0	11.0	0.0	9.8	10.7	10.3
	31	"14,256"	0	0	0	1	5	7	0.0	0.0	0.0	6.5	6.5	6.5
	32	"1,112"	0	0	0	2	3	5	0.0	0.0	0.0	4.0	4.0	4.0
	33	"1,200"	0	0	0	5	4	8	0.0	0.0	0.0	5.7	5.7	5.7
	105	"1,741,336"	2	1	1	10	2	12	11.0	10.9	2.0	8.4	2.2	8.1
	110	"527,656"	4	4	0	1	4	8	6.0	6.0	0.0	4.0	5.8	5.3
	199	"877,504"	16	4	4	1	2	20	6.0	10.7	4.0	6.9	4.1	6.9
	110	"884,584"	2	4	1	1	4	11	5.0	9.0	8.0	7.8	8.0	7.5
	120	"668,688"	37	1	38	1	4	42	4.5	18.1	20.6	6.8	20.3	12.5

Table-19 Analysis of Router Configurations

filename	acl_nr	sum [bytes]	list TCP	list UDP	list ETCP	list IP	min	max	mean TCP	mean UDP	mean ETCP	mean IP	expected case	adversary case
	110	"936,096"	3	4	2	1	4	11	5.0	9.0	8.0	8.9	8.0	7.7
	120	"708,504"	45	1	37	7	4	50	4.5	18.9	20.1	8.4	19.9	13.0
	102	"630,568"	1	1	0	1	3	7	6.1	6.1	0.0	3.0	5.8	5.1
	101	"610,264"	1	1	0	1	3	7	6.7	6.7	0.0	3.0	6.3	5.4
	199	"1,104"	0	0	0	1	4	4	0.0	0.0	0.0	4.0	4.0	4.0
	101	"562,808"	1	2	0	2	4	12	5.8	4.0	0.0	10.9	5.8	6.9
	101	"562,808"	1	2	0	2	4	12	5.8	4.0	0.0	10.9	5.8	6.9
	101	"562,808"	1	2	0	2	4	12	5.8	4.0	0.0	10.9	5.6	6.9
	101	"562,808"	1	2	0	2	4	12	5.8	4.0	0.0	10.9	5.6	6.9
	101	"562,808"	1	2	0	2	4	12	5.8	4.0	0.0	10.9	5.6	6.9
	100	"2,105,304"	8	1	0	1	3	9	4.0	7.3	0.0	3.0	5.2	4.8
	101	"1,158,360"	18	1	0	1	3	24	4.0	11.0	0.0	3.0	6.7	6.0
	102	"689,408"	13	1	0	1	3	21	4.0	10.4	0.0	3.0	6.5	5.8
	100	"2,105,304"	8	1	0	1	3	9	4.0	7.3	0.0	3.0	5.2	4.8
	101	"1,158,360"	18	1	0	1	3	24	4.0	11.0	0.0	3.0	6.7	6.0
	102	"689,408"	13	1	0	1	3	21	4.0	10.4	0.0	3.0	6.5	5.8
	110	"1,249,872"	5	1	0	1	4	13	4.0	10.4	0.0	10.3	7.2	8.2
	111	"532,688"	1	1	0	1	3	4	4.0	4.0	0.0	3.0	3.9	3.7
	112	"529,592"	1	1	0	1	3	4	4.0	4.0	0.0	3.0	3.9	3.7
	113	"534,784"	1	1	0	1	3	4	4.0	4.0	0.0	3.0	3.9	3.7
	114	"603,368"	1	1	0	1	4	11	4.7	8.8	0.0	7.0	6.5	6.8
	114	"603,368"	1	1	0	1	4	11	4.7	8.8	0.0	7.0	6.5	6.8
	115	"533,784"	1	1	0	1	3	4	4.0	4.0	0.0	3.0	3.9	3.7
	116	"533,784"	1	1	0	1	3	4	4.0	4.0	0.0	3.0	3.9	3.7
	101	"284,352"	2	0	0	1	1	12	0.0	6.3	1.0	4.0	1.0	3.8
	102	"1,773,248"	0	0	0	1	5	11	0.0	0.0	0.0	11.0	11.0	11.0
	3	"1,160"	0	0	0	4	3	7	0.0	0.0	0.0	5.0	5.0	5.0
	101	"110,704"	0	0	0	3	5	11	0.0	0.0	0.0	9.8	9.8	9.8
	105	"1,128"	0	0	0	2	3	5	0.0	0.0	0.0	4.3	4.3	4.3
	110	"653,784"	0	0	0	21	5	8	0.0	0.0	0.0	6.9	6.9	6.9
	120	"124,296"	0	0	0	2	5	10	0.0	0.0	0.0	9.5	9.5	9.5
	100	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	101	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	102	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	103	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	104	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	105	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	106	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	107	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	108	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	109	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	110	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	111	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	112	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	113	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	114	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	115	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	116	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	117	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	118	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3

Table-19 Analysis of Router Configurations

filename	acl_nr	sum [bytes]fi	list TCP	list UDP	list ETCP	list IP	min	max	mean TCP	mean UDP	mean ETCP	mean IP	expected case	adversary case
	119	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	120	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	121	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	122	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	123	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	124	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	125	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	126	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	127	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	128	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	129	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	130	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	131	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	132	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	133	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	134	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	135	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	136	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	137	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	138	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	139	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	140	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	141	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	142	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	143	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	144	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	145	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	146	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	147	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	148	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	149	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	101	"1,833,016"	1	1	1	1	2	12	10.6	9.0	2.5	9.5	2.6	7.9
	102	"2,002,288"	1	1	1	1	3	12	10.7	7.6	6.9	8.6	7.0	8.4
	103	"1,425,432"	1	1	1	4	4	13	8.9	9.6	7.3	10.7	7.3	9.1
	104	"2,750,112"	1	2	1	1	3	12	10.7	10.2	9.6	10.4	9.6	10.2
	107	"955,072"	1	2	1	1	3	12	10.7	9.8	9.4	3.0	9.4	8.2
	111	"975,328"	1	1	5	1	2	11	9.7	9.3	4.5	10.0	4.6	8.4
	112	"738,544"	1	1	1	1	2	12	4.3	10.6	2.5	10.5	2.6	7.0
	101	"1,368,816"	1	1	3	1	2	11	10.7	10.2	3.5	7.8	3.6	8.0
	103	"1,202,328"	2	1	1	1	3	11	10.9	11.0	9.8	10.4	9.9	10.5
	105	"782,392"	1	1	1	1	3	11	11.0	11.0	9.9	10.4	9.9	10.6
	107	"2,632,968"	1	0	0	1	3	11	0.0	10.9	0.0	7.5	10.2	9.2
	109	"992,680"	1	1	1	1	3	11	11.0	11.0	10.0	10.3	10.0	10.6
	100	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	101	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	102	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	103	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	104	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	105	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	106	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3

Table-19 Analysis of Router Configurations

filename	acl_nr	sum [bytes]fi	list TCP	list UDP	list ETCP	list IP	min	max	mean TCP	mean UDP	mean ETCP	mean IP	expected case	adversary case
	107	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	108	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	109	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	110	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	111	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	112	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	113	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	114	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	115	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	116	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	117	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	118	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	119	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	120	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	121	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	122	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	123	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	124	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	125	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	126	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	127	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	128	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	129	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	130	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	131	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	132	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	133	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	134	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	135	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	136	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	137	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	138	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	139	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	140	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	141	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	142	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	143	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	144	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	145	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	146	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	147	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	148	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	149	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	101	"1,833,016"	1	1	1	1	2	12	10.6	9.0	2.5	9.5	2.6	7.9
	102	"2,002,288"	1	1	1	1	3	12	10.7	7.8	6.9	8.6	7.0	8.4
	103	"1,425,432"	1	1	1	4	4	13	8.9	9.6	7.3	10.7	7.3	9.1
	104	"2,750,112"	1	2	1	1	3	12	10.7	10.2	9.6	10.4	9.6	10.2
	107	"955,072"	1	2	1	1	3	12	10.7	9.6	9.4	3.0	9.4	8.2
	111	"975,328"	1	1	5	1	2	11	9.7	9.3	4.5	10.0	4.6	8.4
	112	"738,544"	1	1	1	1	2	12	4.3	10.6	2.5	10.5	2.6	7.0

Table-19 Analysis of Router Configurations

filename	acl_nr	sum [bytes]fi	list TCP	list UDP	list ETCP	list IP	min	max	mean TCP	mean UDP	mean ETCP	mean IP	expected case	adversary case
	100	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	101	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	102	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	103	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	104	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	105	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	106	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	107	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	108	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	109	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	110	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	111	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	112	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	113	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	114	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	115	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	116	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	117	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	118	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	119	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	120	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	121	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	122	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	123	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	124	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	125	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	126	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	127	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	128	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	129	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	130	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	131	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	132	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	133	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	134	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	135	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	136	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	137	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	138	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	139	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	140	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	141	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	142	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	143	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	144	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	145	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	146	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	147	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	148	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3
	149	"1,128"	0	0	0	2	3	6	0.0	0.0	0.0	4.3	4.3	4.3

Table-19 Analysis of Router Configurations

filename	acl_nr	sum [bytes]fi	list TCP	list UDP	list ETCP	list IP	min	max	mean TCP	mean UDP	mean ETCP	mean IP	expected case	adversary case
	3	"1,160"	0	0	0	4	3	7	0.0	0.0	0.0	5.0	5.0	5.0
	101	"110,704"	0	0	0	3	5	11	0.0	0.0	0.0	9.8	9.8	9.8
	105	"1,128"	0	0	0	2	3	5	0.0	0.0	0.0	4.3	4.3	4.3
	110	"653,784"	0	0	0	21	5	8	0.0	0.0	0.0	6.9	6.9	6.9
	120	"124,296"	0	0	0	2	5	10	0.0	0.0	0.0	9.5	9.5	9.5
	102	"649,168"	1	1	2	5	2	11	11.0	10.3	3.0	5.5	3.1	7.5
	103	"738,376"	1	1	2	5	2	11	11.0	9.7	3.0	5.5	3.1	7.3
	104	"610,992"	1	1	2	5	2	11	11.0	10.3	3.0	5.5	3.1	7.5
	105	"783,960"	1	1	2	1	2	11	11.0	9.8	3.0	9.0	3.1	8.2
	100	"349,816"	0	0	0	1	3	11	0.0	0.0	0.0	10.9	10.9	10.9
	101	"82,320"	0	0	0	1	3	11	0.0	0.0	0.0	10.5	10.5	10.5
	102	"39,144"	0	0	0	1	3	11	0.0	0.0	0.0	10.1	10.1	10.1
	103	"39,144"	0	0	0	1	3	11	0.0	0.0	0.0	10.1	10.1	10.1
	104	"42,264"	0	0	0	1	3	11	0.0	0.0	0.0	10.0	10.0	10.0
	105	"696,328"	2	1	1	1	2	11	4.0	8.5	2.0	10.8	2.1	6.3
	106	"100,968"	0	0	0	1	3	11	0.0	0.0	0.0	10.7	10.7	10.7
	107	"43,384"	0	0	0	1	3	11	0.0	0.0	0.0	10.5	10.5	10.5
	108	"85,440"	0	0	0	1	3	11	0.0	0.0	0.0	10.8	10.8	10.8
	109	"58,744"	0	0	0	1	3	11	0.0	0.0	0.0	10.5	10.5	10.5
	100	"251,024"	0	0	0	1	11	11	0.0	0.0	0.0	11.0	11.0	11.0
	101	"132,904"	0	0	0	1	3	11	0.0	0.0	0.0	10.8	10.8	10.8
	102	"74,152"	0	0	0	1	3	11	0.0	0.0	0.0	10.5	10.5	10.5
	103	"194,656"	0	0	0	1	3	11	0.0	0.0	0.0	10.8	10.8	10.8
	104	"281,208"	0	0	0	1	3	11	0.0	0.0	0.0	10.9	10.9	10.9
	105	"25,856"	0	0	0	1	3	11	0.0	0.0	0.0	9.8	9.8	9.8
	106	"338,360"	0	0	0	1	3	11	0.0	0.0	0.0	10.9	10.9	10.9
	107	"86,464"	0	0	0	1	3	11	0.0	0.0	0.0	10.8	10.8	10.8
	108	"98,592"	0	0	0	1	3	11	0.0	0.0	0.0	10.9	10.9	10.9
	109	"181,952"	0	0	0	1	3	11	0.0	0.0	0.0	10.2	10.2	10.2
	110	"152,248"	0	0	0	1	3	11	0.0	0.0	0.0	10.8	10.8	10.8
	111	"32,952"	0	0	0	1	3	11	0.0	0.0	0.0	9.9	9.9	9.9
	112	"685,808"	1	0	0	1	3	11	0.0	11.0	0.0	10.8	10.9	10.8
	113	"58,696"	0	0	0	1	3	11	0.0	0.0	0.0	10.4	10.4	10.4
	114	"34,288"	0	0	0	1	3	11	0.0	0.0	0.0	10.8	10.8	10.8
	115	"64,984"	0	0	0	1	3	11	0.0	0.0	0.0	10.8	10.8	10.8
	116	"85,440"	0	0	0	1	3	11	0.0	0.0	0.0	10.8	10.8	10.8
	100	"82,320"	0	0	0	1	3	11	0.0	0.0	0.0	10.5	10.5	10.5
	101	"216,184"	0	0	0	1	3	11	0.0	0.0	0.0	10.8	10.8	10.8
	101	"878,416"	5	7	0	3	1	13	9.8	9.1	1.0	6.4	1.2	6.6
	102	"1,088"	0	0	0	1	3	3	0.0	0.0	0.0	3.0	3.0	3.0
	121	"534,376"	2	2	1	2	2	5	4.5	4.7	2.0	4.3	2.1	3.9
	122	"265,488"	0	1	0	1	3	5	4.5	0.0	0.0	3.0	4.3	3.8
	131	"533,224"	1	2	1	2	2	5	4.5	5.0	2.0	4.3	2.1	3.9
	132	"265,488"	0	1	0	1	3	5	4.5	0.0	0.0	3.0	4.3	3.8
	143	"573,696"	1	2	1	3	2	11	7.6	4.5	2.0	4.6	2.1	4.7
	144	"265,408"	0	1	0	1	3	6	5.0	0.0	0.0	3.0	4.7	4.0
	151	"665,064"	1	2	1	3	2	11	9.3	5.0	2.5	4.6	2.6	5.4
	152	"265,408"	0	1	0	1	3	6	5.0	0.0	0.0	3.0	4.7	4.0
	161	"705,840"	1	2	0	3	1	11	6.4	9.1	1.0	4.6	1.1	5.3
	162	"294,704"	0	1	0	1	3	11	10.6	0.0	0.0	3.0	9.3	6.8

Table-19 Analysis of Router Configurations

filename	acl_nr	sum [bytes]ll	list TCP	list UDP	list ETCP	list IP	min	max	mean TCP	mean UDP	mean ETCP	mean IP	expected case	adversary case
	163	"603,680"	1	2	0	2	1	11	8.4	10.6	1.0	4.3	1.2	6.1
	164	"265,408"	0	1	0	1	3	6	5.0	0.0	0.0	3.0	4.7	4.0
	171	"621,608"	3	3	0	2	1	11	8.3	5.0	1.0	4.3	1.1	4.6
	121	"897,456"	3	7	0	3	1	13	9.8	9.0	1.0	6.4	1.2	6.6
	122	"1,088"	0	0	0	1	3	3	0.0	0.0	0.0	3.0	3.0	3.0
	161	"608,832"	1	2	1	1	2	11	9.0	4.5	2.0	3.0	2.1	4.6
	162	"265,536"	0	2	0	1	3	5	4.7	0.0	0.0	3.0	4.4	3.8
	171	"534,392"	2	2	1	1	2	5	4.6	4.4	2.5	3.0	2.5	3.6
	172	"265,488"	0	1	0	1	3	5	4.5	0.0	0.0	3.0	4.3	3.8
	141	"780,680"	1	2	0	5	1	11	7.8	9.1	1.0	6.1	1.1	6.0
	142	"767,440"	1	1	0	1	1	11	8.7	7.5	1.0	3.0	1.1	5.1
	141	"897,344"	2	4	0	1	1	11	8.8	7.5	1.0	8.7	1.1	6.5
	102	"637,456"	2	1	0	2	1	10	5.1	7.2	1.0	6.2	1.1	4.9
	142	"1,148,168"	4	10	0	18	1	21	7.2	8.1	1.0	9.5	1.1	6.4
	102	"789,256"	3	4	0	2	1	11	8.4	7.9	1.0	6.0	1.1	5.8
	112	"552,560"	1	1	0	1	3	6	5.0	6.0	0.0	3.0	5.2	4.7
	122	"556,848"	2	2	0	3	1	7	4.8	6.0	1.0	5.5	1.1	4.3
	141	"897,344"	2	4	0	1	1	11	8.8	7.5	1.0	8.7	1.1	6.5
	131	"3,559,528"	2	4	0	1	1	11	9.8	8.7	1.0	9.4	1.2	7.2
	141	"5,058,208"	2	4	0	3	1	11	8.3	6.7	1.0	5.9	1.1	5.5
	141	"845,360"	2	4	0	1	1	11	8.6	7.3	1.0	8.7	1.1	6.4
	101	"1,149,800"	2	27	0	4	1	33	13.7	9.0	1.0	5.0	1.2	7.2
	102	"1,088"	0	0	0	1	3	3	0.0	0.0	0.0	3.0	3.0	3.0
	111	"602,664"	1	2	1	3	2	11	7.9	5.0	2.0	4.6	2.1	4.9
	112	"546,944"	3	2	0	1	3	11	7.3	5.0	0.0	3.0	5.9	5.1
	122	"265,464"	0	1	0	1	3	5	4.5	0.0	0.0	3.0	4.3	3.8
	131	"533,224"	1	2	1	2	2	5	4.5	5.0	2.0	4.3	2.1	3.9
	132	"265,464"	0	1	0	1	3	5	4.5	0.0	0.0	3.0	4.3	3.8
	151	"665,040"	1	2	1	2	2	11	9.3	5.0	2.5	4.3	2.6	5.3
	152	"265,408"	0	1	0	1	3	6	5.0	0.0	0.0	3.0	4.7	4.0
	121	"1,160,752"	2	27	0	3	1	33	13.8	9.0	1.0	6.4	1.2	7.6
	122	"1,064"	0	0	0	1	3	3	0.0	0.0	0.0	3.0	3.0	3.0
	132	"547,008"	3	2	0	1	3	11	8.0	5.0	0.0	3.0	6.3	5.3
	141	"592,284"	3	2	0	2	1	11	9.1	4.8	1.0	4.3	1.1	4.8
	142	"31,432"	0	0	0	1	3	11	0.0	0.0	0.0	8.2	8.2	8.2
	161	"586,424"	1	2	1	1	2	11	8.8	4.5	2.0	3.0	2.1	4.6
	162	"265,536"	0	2	0	1	3	5	4.7	0.0	0.0	3.0	4.4	3.8
	102	"814,504"	1	2	0	4	3	11	9.0	9.4	0.0	4.8	8.8	7.8
	112	"549,408"	1	1	0	1	3	6	5.0	6.0	0.0	3.0	5.2	4.7
	131	"2,428,024"	2	4	0	1	1	11	9.5	7.6	1.0	9.6	1.2	6.9
	101	"1,179,992"	2	28	0	3	1	34	13.9	9.0	1.0	6.4	1.2	7.6
	102	"1,088"	0	0	0	1	3	3	0.0	0.0	0.0	3.0	3.0	3.0
	111	"602,664"	1	2	1	3	2	11	7.9	5.0	2.0	4.6	2.1	4.9
	112	"547,008"	3	2	0	1	3	11	7.7	5.0	0.0	3.0	6.1	5.2
	122	"265,488"	0	1	0	1	3	5	4.5	0.0	0.0	3.0	4.3	3.8
	131	"533,224"	1	2	1	2	2	5	4.5	5.0	2.0	4.3	2.1	3.9
	132	"265,488"	0	1	0	1	3	5	4.5	0.0	0.0	3.0	4.3	3.8
	151	"665,064"	1	2	1	3	2	11	9.3	5.0	2.5	4.6	2.6	5.4
	152	"265,408"	0	1	0	1	3	6	5.0	0.0	0.0	3.0	4.7	4.0
	121	"1,179,992"	2	28	0	3	1	34	13.9	9.0	1.0	6.4	1.2	7.6

Table-19 Analysis of Router Configurations

filename	acl_nr	sum [bytes]fi	list TCP	list UDP	list ETCP	list IP	min	max	mean TCP	mean UDP	mean ETCP	mean IP	expected case	adversary case
	122	"1,088"	0	0	0	1	3	3	0.0	0.0	0.0	3.0	3.0	3.0
	132	"547,008"	3	2	0	1	3	11	7.7	5.0	0.0	3.0	6.1	5.2
	141	"592,328"	3	2	0	2	1	11	9.2	4.8	1.0	4.3	1.1	4.8
	142	"31,432"	0	0	0	1	3	11	0.0	0.0	0.0	8.2	8.2	8.2
	161	"608,832"	1	2	1	1	2	11	9.0	4.5	2.0	3.0	2.1	4.8
	162	"265,536"	0	2	0	1	3	5	4.7	0.0	0.0	3.0	4.4	3.8
	141	"699,496"	1	2	0	5	1	11	7.7	8.2	1.0	6.1	1.1	5.8
	142	"1,088"	0	0	0	1	3	3	0.0	0.0	0.0	3.0	3.0	3.0
	141	"677,504"	2	2	0	1	1	11	7.3	7.5	1.0	6.4	1.1	5.5
	102	"832,056"	1	2	0	4	1	11	9.0	9.5	1.0	4.8	1.2	6.1
	112	"550,472"	1	1	0	1	3	8	5.0	6.0	0.0	3.0	5.2	4.7
	141	"677,504"	2	2	0	1	1	11	7.3	7.5	1.0	6.4	1.1	5.5
	131	"2,592,960"	2	4	0	1	1	11	9.5	7.5	1.0	9.7	1.2	6.9
	100	"1,114,392"	367	3	1	1	2	374	9.2	102.2	2.5	7.6	3.4	30.4
	101	"830,360"	14	1	2	1	2	19	10.1	8.2	3.0	3.5	3.1	6.2
	101	"567,584"	2	1	0	0	4	8	4.0	6.7	0.0	0.0	5.2	5.4
	102	"38,400"	0	0	0	1	5	8	0.0	0.0	0.0	6.1	6.1	6.1
	101	"264,296"	1	0	0	1	3	4	0.0	4.0	0.0	3.5	3.9	3.8
	102	"769,832"	6	1	0	9	3	13	4.5	9.5	0.0	8.8	6.9	7.6
	103	"674,488"	1	1	0	1	3	10	4.0	7.7	0.0	8.5	5.9	6.8
	101	"343,464"	1	0	0	1	3	10	0.0	10.0	0.0	3.0	8.6	6.5
	102	"880,040"	8	1	0	21	3	26	4.5	9.9	0.0	12.4	7.5	9.0
	150	"421,696"	1	0	0	1	3	10	0.0	9.5	0.0	9.0	9.4	9.3
	101	"264,296"	1	0	0	1	3	4	0.0	4.0	0.0	3.5	3.9	3.8
	102	"880,040"	8	1	0	21	3	26	4.5	9.9	0.0	12.4	7.5	9.0
	103	"775,280"	1	1	0	2	3	11	4.0	7.7	0.0	9.8	6.1	7.2
	101	"56,648"	0	0	0	1	3	11	0.0	0.0	0.0	10.4	10.4	10.4
	102	"18,904"	0	0	0	1	3	9	0.0	0.0	0.0	8.7	8.7	8.7
	106	"265,336"	0	1	0	1	3	5	5.0	0.0	0.0	3.0	4.7	4.0
	100	"12,976"	0	0	0	2	5	8	0.0	0.0	0.0	6.3	6.3	6.3
	101	"25,408"	0	0	0	1	5	11	0.0	0.0	0.0	6.5	6.5	6.5
	101.	"35,008"	0	0	0	1	9	11	0.0	0.0	0.0	10.0	10.0	10.0
	102	"29,952"	0	0	0	1	9	11	0.0	0.0	0.0	9.8	9.8	9.8
	100	"12,976"	0	0	0	2	5	8	0.0	0.0	0.0	6.3	6.3	6.3
	101	"25,408"	0	0	0	1	5	11	0.0	0.0	0.0	6.5	6.5	6.5
	101	"35,008"	0	0	0	1	9	11	0.0	0.0	0.0	10.0	10.0	10.0
	102	"29,952"	0	0	0	1	9	11	0.0	0.0	0.0	9.8	9.8	9.8
	100	"12,976"	0	0	0	2	5	8	0.0	0.0	0.0	6.3	6.3	6.3
	101	"25,408"	0	0	0	1	5	11	0.0	0.0	0.0	6.5	6.5	6.5
	101	"35,008"	0	0	0	1	9	11	0.0	0.0	0.0	10.0	10.0	10.0
	102	"29,952"	0	0	0	1	9	11	0.0	0.0	0.0	9.8	9.8	9.8

Table-19 Analysis of Router Configurations

Appendix G: References

[1] Cisco IOS Software Command Summary

[2] Cisco IOS Security Configuration Guide

[3] ACL Processing in Hardware, Dominique Alessandri, Oct 1997, Diploma Thesis ETH

[4] The Art of Computer Programming, Donald Knuth, 2nd ed. 1973, Addison-Wesley

[5] Computer Networks, Andrew Tanenbaum, 3rd ed. 1996, Prentice Hall

[6] Class Construction in C and C++, Roger Sessions, 1992, Prentice Hall

[7] Effective C++, Scott Meyers, 1992, Addison-Wesley

[8] More Effective C++, Scott Meyers, 1996, Addison-Wesley